

Towards Formal Verification of Program Obfuscation

Weiyun Lu*, Bahman Sistany[†], Amy Felty*[‡], Philip Scott^{‡*}

*School of Electrical Engineering and Computer Science, University of Ottawa, Canada

Email: {wlu058,afelty,philip.scott}@uottawa.ca

[†]Cloakware Research Irdeto Canada, Ottawa, Canada

Email: bahman.sistany@irdeto.com

[‡]Department of Mathematics and Statistics, University of Ottawa, Canada

Abstract—Code obfuscation involves transforming a program to a new version that performs the same computation but hides the functionality of the original code. An important property of such a transformation is that it preserves the behavior of the original program. In this paper, we lay the foundation for studying and reasoning about code obfuscating transformations, and show how the preservation of certain behaviours may be formally verified. To this end, we apply techniques of formal specification and verification using the Coq Proof Assistant. We use and extend an existing encoding of a simple imperative language in Coq along with an encoding of Hoare logic for reasoning about this language. We formulate what it means for a program’s semantics to be preserved by an obfuscating transformation, and give formal machine-checked proofs that these behaviours or properties hold. We also define a lower-level flowchart language which is “wrapped around” our imperative language, allowing us to model certain flattening transformations and treat blocks of codes as objects in their own right.

Index Terms—obfuscation, verification, security, correctness, Coq, proof

1. Introduction

We expect our software systems to function correctly. By “correctly”, we mean that a system will behave according to explicit and/or implicit expectations, i.e., its written and/or unwritten specifications. Typically, extensive testing is done to increase the confidence in the correct functionality of a piece of software. The more testing that is done, the more confidence one has of the likelihood of correctness, but positive testing results are not a proof of correctness. In systems where more assurance of correctness is required, various types of deductive reasoning are often used, where formal verification methods based on theoretical foundations rooted in logic are employed. This is the approach that we adopt here. It is important to note that formal verification transfers the problem of confidence in program correctness to the problem of confidence in specification correctness. Thus, it is not a silver bullet; however, since specifications are often smaller and less complex to express, we are able to successfully reduce the trusted computing base (TCB) and increase our chances of achieving correctness.

A common approach to formal verification used to show a (software) system behaves according to its specification is to employ a specification language based on

the familiar use of *assertions* [1], [2]. A specification (in the form of a statement about program fragments that are expected to hold) is typically expressed in some variation of first-order logic, and the verification system will try to either prove the assertions correct or signal that they don’t hold. This can be a rather elaborate process. For example, assertions can be used to generate logical formulas called *verification conditions* (VCs), which are either fed into a *satisfiability modulo theories* (SMT) solver behind the scenes in a verification backend, or are presented in a more visible manner to a verification expert who will manually discharge them. VC generation for program verification goes back to at least Hoare logic [2], Eiffel style contracts [3], and proof-carrying-code (PCC) [4].

In this paper, we use a formal verification approach to the task of program transformations. In particular, the question we address here is: how can one perform obfuscating or optimizing transformations on code that is assumed to be correct with respect to certain specified behaviour (expressed in some assertion language) while preserving the correctness of the specified behaviour?

To achieve the highest level of assurance that a transformation has maintained correctness of the code, one should prove that the two versions of the program (before and after the transformation) are equivalent. Although there exist large-scale verification results such as the formally verified compiler CompCert [5], in general scaling up verification efforts to realistic systems is extremely hard. CompCert involved the verification of a semantic equivalence between C and a generated assembly language, and took several man-years to complete.

Equivalence between two versions of a program certainly implies the correctness of any program properties of interest proved for one version applies to the other. Alternatively, what if we limit ourselves to only proving properties of interest in the “before” version of a program, and show that these properties are maintained in the “after” version of the program after a particular transformation is applied? Here, our focus is on certain simple transformations that don’t invalidate properties about the “before” version versus the “after” version. For example, consider the program snippet in listing [1], which asserts that $y > 2$ and is easily verified visually. The snippet in listing [2] illustrates a simple obfuscating transformation called *variable splitting*, where we have split the variable x into two variables $x1$ and $x2$, and it is clear (visually) that the assertion $y > 2$ still holds.

In general, though, most transformations, whether op-

```
x = 2; y = 5;
y = x + y;
assert(y > 2);
```

Listing 1: Original Code

```
x1 = 1; x2 = 1; y = 5;
y = x1 + y; y = x2 + y;
assert(y > 2);
```

Listing 2: Obfuscated Code

```
int main (int argc, char *argv[])
{
    unsigned int x = 10;
    unsigned int y = 20;
    unsigned int z = 0;

    z = x + y;
    assert(z == 30);
    return 0;
}
```

Listing 3: Original Code

```
int main (int argc, char *argv[])
{
    unsigned int x = 10;
    unsigned int y = 20;
    unsigned int z = 0;

    unsigned int a = ((unsigned int)argc);
    unsigned int w = a * a;

    w = a + w;
    w = w % 2;
    if (w == 0)
    {
        z = x + y;
    }
    else
    {
        z = y - x;
    }
    assert(z == 30);
    return 0;
}
```

Listing 4: Obfuscated Code

timizations or obfuscations, invalidate assertions that hold true about the “before” version. Obfuscation is particularly troublesome, because its main goal is to hide the functionality of the code from prying eyes while maintaining the functionality of the “before” program. More formally, according to Barak et al. as cited in [6], obfuscators are programs that transform an input program (e.g. similar to a compiler) into an output program such that the output program satisfies the following three properties:

- 1) it is semantically equivalent to the input program (*functionality property*);
- 2) it is at most polynomially bigger or slower than the input program (*slowdown property*) and
- 3) it is as “hard to analyze and de-obfuscate” as a blackbox version of the program (*virtual black-box property*).

“Prying eyes” could, for example, be some kind of static analysis tool where an attacker is trying to determine certain facts about the code, and obfuscation is trying to make this difficult. The program in listing 3 is correct with respect to the assertion that is expressed (e.g. $z == 30$) as is evident by simple inspection of the code. The program snippet in listing 4 is the “after” program where a non-linear *opaque predicate transformation* (see Section 3) has been applied to hide the fact that at program’s end, the value of z is in fact 30. In this case, it follows from the fact that $\forall x \in \mathbb{Z}, ((x^2 + x) \bmod 2) == 0$. This paper describes steps towards implementing a framework in the Coq Proof Assistant [7] for a simple imperative language that allows us to study obfuscating transformations, their impact on programs, and how specified behaviour may be preserved beyond the transformations. Our starting point is the IMP language from [8], which includes an encoding in Coq of a familiar small imperative language along with its formalized semantics. A number of initial goals and principles drove the direction of this research: 1) We don’t want to reinvent the wheel, which is why we start with IMP. 2) We want to assure accessibility to as wide an audience as possible. For this reason, we choose IMP over CompCert and Clight, which are used in [9]. On the one hand, building on CompCert would have given us lots of proofs and formalisms for free; however, the significant learning curve associated with this infrastructure seemed prohibitive, and thus much less accessible. 3) We want the framework to be extendable. Following the lead of [8], where a number of extensions to IMP are easily added and studied, we wanted the ability to build our obfuscation infrastructure incrementally on top of IMP.

Keeping these research goals in mind the contributions

of this paper are the following:

- We consider different formulations of what it means for a transformation to be semantics-preserving: we cover *command equivalence* in Sections 2.2 and 3.1 as well as *Hoare logic equivalence* in Section 3.2. By command equivalence, we mean that if the two programs start out in the same state, then the state resulting after execution of each program is exactly the same. By state, we mean variable names and their values; these values can change during execution, but at the end, all variable values must have changed in exactly the same way. This is the notion of equivalence mentioned above. Hoare logic equivalence is a weaker notion that is enough for most code obfuscation transformations. It allows an obfuscated program to use and modify variables that don’t occur in the assertion. Only the values of variables that occur in the assertion are required to have the same values at the end of execution. Using Hoare logic equivalence is a novel approach, which leads to establishing our main strategy to “Formal Verification of Program Obfuscation”: our obfuscated programs will be “decorated” à la Pierce [8] with additional assertions whose proofs will also be provided. The

additional assertions that our obfuscators will provide will be needed to show the original desired behaviour still holds after an obfuscation.¹

- We give motivations and top-level explanations for the proofs² (and refer to [10] for a more detailed treatment of the proofs and tactics) in Coq, which, to the best of our knowledge, the existing literature does not, thus providing an accessible explanation of not just obfuscation techniques, but also in tandem with its formalization and verification inside Coq. This follows from research goal 2.³
- We begin with a minimal version of IMP and augment it as needed for *control flow flattening algorithms* (see Section 4), first by augmenting its syntax and semantics with switch statements, and then by defining a lower-level flowchart language that wraps around blocks of code in order to model real-world intermediate languages used in obfuscation tools. This follows our research goals 2 and 3.

This project, and in particular, the formalization in Coq, was motivated by the Cloakware obfuscation tool produced at Irdeto, and the interest in showing the correctness of some of its core functionality in safety critical environments.⁴ For details of the Coq code and fuller explanations of the proofs, the reader is referred [?] and [10], respectively.^{5,6}

2. The IMP Language and Hoare Logic

We now give the necessary definitions and theorems from Software Foundations [8] needed to understand our formalization in the sections that follow. Along the way, we describe the features of Coq that we use. For more details on IMP, the reader is referred to [8].

2.1. Syntax and Semantics of IMP

Informally, IMP commands are defined by the following BNF grammar: $c ::= \text{SKIP} \mid x ::= a \mid c ; c \mid \text{IFB } b \text{ THEN } c \text{ ELSE } c \text{ FI} \mid \text{WHILE } b \text{ DO } c \text{ END}$. In [8], this grammar is encoded by defining a new type called `com` and new keywords (called *constructors* in Coq), one for each element of the grammar, and then defining notation that maps this formal definition to the BNF syntax. For example, the definition of `com` includes the following case for assignment statements: `CAss (x : string) (a :`

1. Bahman and Amy: find a good place to cover this proposal and if so update the reference to 3.2 above.

2. AF: We agreed to remove this paragraph. I am leaving it here for now until we figure out where to refer to Wei's thesis, and maybe include a discussion about this at that point.

3. Phil asks Why?

4. Much of it was carried out during the first author's co-op term at Irdeto under the supervision of the second author, who further developed it in his Master's thesis [10]; all lemmas and theorems referred to in this paper are formally proved in Coq.

5. Bahman: just marking references to Wei's work. At the end, some may be redundant and/or out of place.

6. AF: I moved this text up from Section 2 to Section 1, and I think we can remove many/most of the other references to the Coq code and the proofs (which I have done). I think this addresses Phil's comments (top of page 1 and the bottom of page 2 on his annotated pdf, and comment 2 in his email).

`aexp)`, which introduces the `CAss` constructor and specifies its two arguments along with their types. A variable is represented using the built-in `string` type in Coq, the arithmetic expression on the right of an assignment command has type `aexp`, and the notation specifies that Coq terms of the form `(CAss x a)` will be written `x := a`. The definition of `aexp` in [8] is defined using the same technique—starting from a BNF grammar, constructors are defined and then notation is given so that the syntax matches the grammar. We omit its definition as well as the definition of `bexp` for representing boolean expressions, which appear in `IFB` and `WHILE` commands.

To define the operational semantics of IMP, the notion of a state is needed. We introduce a type called `state` defined to be an abbreviation for the type `string -> nat`. Here, `nat` is Coq's built in type for natural numbers \mathbb{N} . Although, we restrict our formal development to natural numbers, it could be easily extended to other types such as `int` or `float`. An *initial state* in Coq is a function that maps all variables to the default value 0. States are modified using an update operator. For example, the state `s & { X --> 3; Y --> 1 }` represents a state in which `s` is modified so that `X` has the value 3 and `Y` has the value 1. The “`s &`” can be omitted if the update is to an initial state.

Evaluation of commands can be viewed as a function from a command and a starting state to a new state. It is encoded as a relation in Coq, and the notation `c / s1 \\\ s2` means that starting in state `s1` and executing `c` results in a new state `s2`. We say that “`c` takes `s1` to `s2`.” The definition of this relation relies on two functions. The first is `aeval`; the expression `(aeval s a)` evaluates `a` (an element of `aexp`) in a state `s`, resulting in a natural number. The function `beval` is similar and maps a `bexp` to a Coq boolean. (We omit the details of these definitions.)

2.2. Command Equivalence Definition

For two commands (IMP programs) c_1 and c_2 to be command equivalent means that for any pair of states st and st' , c_1 takes st to st' if and only if c_2 takes st to st' . In Coq, `cequiv` [8] is defined as a predicate that takes two commands as arguments:

Definition 2.1 (Command equivalence).

```
Definition cequiv (c1 c2 : com) : Prop
:= forall (st st' : state),
(c1 / st \\\ st') <-> (c2 / st \\\ st').
```

`Prop` is the type of logical formulas in Coq, and `forall` and `<->` are the notations for universal quantification and logical equivalence, respectively. The corresponding definitions for arithmetic expressions and booleans are also important.

Definition 2.2 (Arithmetic/boolean equivalence).

```
Definition aequiv (a1 a2 : aexp) : Prop
:= forall s, aeval s a1 = aeval s a2.
Definition bequiv (b1 b2 : bexp) : Prop
:= forall s, beval s b1 = beval s b2.
```

2.3. Hoare Logic

Hoare logic is a way for us to prove that executing a program will result in satisfying certain post-conditions, (possibly) conditional on certain pre-conditions being met. This involves defining a natural way of writing program specifications, along with a compositional proof technique to prove correctness with respect to them.

Definition 2.3 (Hoare triple). A *Hoare triple* is a triple consisting of a pre-condition P , a program c , and a post-condition Q , written

$$(|P|) \ c \ (|Q|),$$

which specifies that whenever P is true before execution, running the program c is guaranteed to make Q true after execution.

This informal definition leaves states implicit, but for the formal definition in Coq we will need to take states into account.

Definition 2.4 (Assertion). An *assertion* about a program's state is a function from states to propositions.

Definition Assertion := state -> Prop.

Informally, for some assertion P and some state st , the proposition $P(st)$ represents the statement that P holds in state st . As an example, let st be the state where the value of every variable is 0. Let P be the assertion that $x = 0$. Then $P(st)$ is the proposition “ $x = 0$ in the state st ”. In Coq, Hoare triples have three arguments, a command and two assertions:

```
Definition hoare_triple (P:Assertion)
  (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st \ st' ->
  P st -> Q st'.
```

and expresses that if P holds in state st and c takes st to st' , then Q holds in st' . We will sometimes use the notation $\{\{P\}\}c\{\{Q\}\}$.

Proving that a Hoare triple holds is a line-by-line affair, starting from the bottom of a program and working upwards. There is one rule for each kind of IMP command, and the application is mostly mechanical, except for the WHILE statement (see [8]).

3. Opaque Predicates in IMP/Coq

An *opaque predicate* [11] is a predicate that always evaluates to either *true* or *false* and the truth-value is known to the transformation but hard to deduce by an attacker [12]. It could be any boolean expression in a program, but we will only be concerned with those involving arithmetic formulas here. An opaque predicate is used, for example, in an IFB statement, and in the case when it evaluates to *true*, the code under the *false* branch is never evaluated at runtime. Thus opaque predicates incur no runtime performance penalty.

Of course, the most basic opaque predicates are just the boolean constants *true* and *false* themselves, but these are not very useful in practice because it is immediately obvious what is happening in the program, and neither the simplest of humans nor tools will be fooled. For a

more advanced treatment of opaque predicates and how they may be broken see [12].

An *opaque predicate transformation* takes as inputs a program to be obfuscated, c_1 , an opaque predicate P , and a dummy program c_2 , and returns the program:

```
IFB (P x) THEN c1 ELSE c2 FI.
```

It is not known to an attacker, a priori, that c_2 is a dummy program. In practice, c_2 should be complicated enough to appear that it could feasibly be intended to be executed. For example, c_2 is often an actual command/statement randomly chosen to be the dummy program.

In Section 3.1, we describe our initial (straightforward, naive) attempt, in which the transformation introduces new variables (as is standard in writing code in a typical imperative language), assigns them values and then uses them in the opaque predicate. Using this approach, trying to state a general theorem about command equivalence ends up being problematic.

We then discuss how this problem spawned two ideas in different directions, which both rectify the issue. First, we keep the above approach to adding variables, values, and an opaque predicate, but use Hoare logic to prove assertions that are weaker than command equivalence (Section 3.2). Second, we reformulate the transformation to rely on values already existing in the state of the program. Here, we assume that the opaque predicate transformation is being applied to a small piece of code somewhere within a much larger program, one where these values already exist in the state (Section 3.3).⁷

Finally in Section 3.4, we again employ Hoare logic and consider an example where we assume that the attacker does not know the constant truth value of the opaque predicate,⁸ but can use static analysis to gain information, and we show that the attacker ends up gaining weaker knowledge.

3.1. Command Equivalence

Definition 3.1 (Factorial program (countdown nonzero formulation)). The following IMP program computes the factorial of a nonzero natural number. The input is read from X , temporary values are stored as Z , and the factorial of the input is stored in Y as the output.

```
Definition fact_nonzero : com :=
  Z ::= X;;
  Y ::= 1;;
  WHILE ! (Z <= 1) DO
    Y ::= Y * Z;;
    Z ::= Z - 1
  END.
```

The choice of factorial program as a candidate for examples of obfuscation is somewhat arbitrary. It works well

7. Bahman: this sentence doesn't read well. Amy could you review?
Amy: Done.

8. Bahman: this is informal and makes sense but should we use proof object terminology? “an attacker who does not have the proof for the opaque predicate”

Phil: I like the way it sounds now—an attacker not knowing its constant truth sounds better to me than just not happening to have a proof.

Amy: I tried to reword this whole paragraph. I hope it is correct. There was a phrase “because of it” in the original, and I didn't know what “it” was referring to. (The original text is still there in comments.) Wei, can you check this?

for illustrative purposes, as it is neither too complex nor completely trivial.

Example 3.2. The *fact_nonzero* program with input $X = 3$ yields output $Y = 6$.

```
Example factorial_3:
fact_nonzero / { X --> 3 } \\\
{ X --> 3; Z --> 3; Y --> 1; Y --> 3;
  Z --> 2; Y --> 6; Z --> 1 }.
```

The `Example` keyword in Coq is followed by a name, a colon, and a statement, which is then followed by a proof. Note that in the above statement, the final state holds the information of every intermediate assignment made by the program. The rightmost value of each variable contains its final value and thus we can discern the correct output $Y = 6$. (For the Coq proof of this statement, and all other examples and theorems in this and the next section, refer to [?].)

For this section, we'll use as a simple opaque predicate namely,

$$\forall x. (x * x + x + x + 1) = (x + 1) * (x + 1).$$

We now define an opaque predicate transformation with our running example. For the purposes of making the proofs easier to work with, and also to add a slight additional touch of obfuscation, we split up these assignments over multiple lines, as follows.

```
Definition opaque_trans x c1 c2 :=
  X' ::= (ANum x) ;;
  Z' ::= X' * X' ;;
  Z' ::= Z' + X' ;;
  Z' ::= Z' + X' ;;
  Z' ::= Z' + 1 ;;
  Z'' ::= X' + 1 ;;
  Z'' ::= Z'' * Z'' ;;
  IFB (BEq Z' Z'') THEN c1 ELSE c2 FI.
```

Here $(\text{ANum } x)$ is the representation of the number x as an arithmetic expression (of type `aexp`). The *opaque_trans* function takes three arguments as input: a number x , and programs c_1 and c_2 , and returns the new program that executes c_1 if the equation

$$(x * x + x + x + 1) = (x + 1) * (x + 1)$$

holds and executes c_2 otherwise. Of course, the above equation is true for all x (which we have proved in Coq), so the resulting program should be the same as c_1 . We'd like to claim that a program transformed by *opaque_trans* is equivalent to the original.

What do we mean when we say the transformed program should be “equivalent”? Example 3.3 shows what happens when *opaque_trans* is applied to the *factorial_3* example.

Example 3.3. For any $x \in \mathbb{N}$ and any program c_2 , *opaque_trans* x *fact_nonzero* c_2 with input $X = 3$ yields output $Y = 6$. In Coq, however, it looks as follows.

```
Example factorial_3_opaque_trans:
forall x c2,
opaque_trans x fact_nonzero c2 /
{ X --> 3 } \\\
{ X --> 3; X' --> x; Z' --> x * x;
```

```
Z' --> x * x + x;
Z' --> x * x + x + x;
Z' --> x * x + x + x + 1;
Z'' --> x + 1;
Z'' --> (x + 1) * (x + 1);
Z --> 3; Y --> 1; Y --> 3;
Z --> 2; Y --> 6; Z --> 1 }.
```

The proof of this statement begins by instantiating variables and unfolding definitions; after which the transformed program becomes:

```
X' ::= x;;
Z' ::= X' * X';;
Z' ::= Z' + X';;
Z' ::= Z' + X';;
Z' ::= Z' + 1;;
Z'' ::= X' + 1;;
Z'' ::= Z'' * Z'';;
IFB Z' = Z''
THEN Z ::= X;; Y ::= 1;;
  WHILE ! (Z <= 1) DO
    Y ::= Y * Z;; Z ::= Z - 1
  END
ELSE c2 FI
```

The proof of this theorem illustrates that our opaque predicate transformation worked; however, there seems to be no direct way to generalize Example 3.3 in terms of command equivalence. We cannot use *cequiv* (Definition 2.1)—that is, we can't use it with the current formulation of the transformation—since the transformation introduces new variables, which affect the value of the end state, even if those variables are not of interest to us. In particular, note that although variable Y has the same value in the end states in Examples 3.2 and 3.3, the rest of the state information is not identical. We'll revisit command equivalence in Section 3.3.

3.2. Hoare Logic Equivalence

In this section, we explore using Hoare logic to specify program conditions, and then generalize the result as much as we can. The main idea with Hoare logic is that we can be more specific about what we wish a transformation to preserve which it turns out is often good enough. First, we'll use a slightly different formulation of the factorial program.

Definition 3.4 (Factorial program (count-up formulation)). This version of the factorial program counts up from zero rather than down from X , and works for input 0 as well.

```
Definition fact_program : com :=
  Y ::= 1;;
  Z ::= 0;;
  WHILE ! (Z = X) DO
    Z ::= Z + 1;;
    Y ::= Y * Z
  END.
```

We begin by defining assertions about the values of X and Y in Coq as follows.

```
Definition as_x (x0 : nat) : Assertion
:= (fun st => st X = x0).
```

```
Definition as_y (y0 : nat) : Assertion
:= (fun st => st Y = y0).
```

We restate Example 3.2, replacing the specific values of 3 and 6 with arbitrary natural numbers, expressed as the following Hoare triple:

Example 3.5. $(|X = x_0|) \text{ fact_program } (|Y = x_0!|)$, which in Coq is:

```
Example factorial_all_hoare: forall xo,
  {{ as_x xo }}
  fact_program
  {{ as_y (fact xo) }}.
```

Example 3.5 states that when $X = x_0$ before the (un-obfuscated) *fact_program* runs, then $Y = x_0!$ afterward, where *fact* above is the mathematical definition of the factorial function expressed in Coq.

We now turn to showing that when we obfuscate *fact_program*, it remains the case that $X = x_0$ beforehand implies that $Y = x_0!$ when the program finishes. In the following, we use a new formulation of the opaque predicate transformation, where we collapse the assignments into single lines.

```
Definition opaque_trans' x c1 c2 :=
  X' ::= (ANum x) ;;
  Z' ::= X' * X' + X' + X' + 1 ;;
  Z'' ::= (X' + 1) * (X' + 1) ;;
  IFB (BEq Z' Z'') THEN c1 ELSE c2 FI.
```

Example 3.6. We now apply the transformation *opaque_trans'* to *fact_program* to obtain the following statement expressing that the same Hoare triple holds with the obfuscated factorial program in place of the original program.⁹

$$\forall x_0, c_2 \\ (|X = x_0|) \text{ opaque_trans}' X \text{ fact_program } c_2 \\ (|Y = x_0!|)$$

which in Coq is:

```
Example factorial_all_hoare_opaque:
forall x xo c2,
  {{ as_x xo }}
  (opaque_trans' x fact_program c2)
  {{ as_y (fact xo) }}.
```

At this point, we've stated that our factorial program, both with and without the opaque predicate transformation, satisfies a Hoare triple of the form

$$(|X = x_0|) c (|Y = y_0|).$$

Our result is still rather specific in the sense that the only pre-condition we treat is that a specific variable X takes on some value, and the only post-condition we treat is that a specific variable Y takes on some value. We now would like to generalize, which we do in two steps. To do so, we introduce a new term *Hoare fidelity*. The first definition introduces the general concept without generalizing the pre- and post-conditions.

Definition 3.7 (Hoare fidelity (with respect to input X and output Y)). A program c_2 preserves the Hoare

9. AF: Using both the Hoare logic notation and the Coq notation is sometimes confusing. Here, we need an explanation about why X is used in the Hoare triple and universally quantified x is used in the Coq code. I don't understand it myself.

fidelity of a program c_1 with respect to input X and output Y , if the validity of the Hoare triple

$$(|X = x_0|) c_1 (|Y = y_0|)$$

implies the validity of the Hoare triple

$$(|X = x_0|) c_2 (|Y = y_0|).$$

In Coq,

```
Definition Hoare_fidelity_xy c1 c2
:= forall xo yo,
  hoare_triple (as_x xo) c1 (as_y yo) ->
  hoare_triple (as_x xo) c2 (as_y yo).
```

Indeed, the decision to use the factorial program in the previous examples for illustrative purposes was an unnecessary detail, so we replace it with an arbitrary program, and prove the following general theorem.

Theorem 3.8. For all programs c_1 and c_2 , and all $x \in \mathbb{N}$, the transformed program $(\text{opaque_trans}' x c_1 c_2)$ preserves the Hoare fidelity of c_1 with respect to input X and output Y . In Coq,

```
Theorem Opaque_trans_hoare_fidelity_xy:
forall x c1 c2,
  Hoare_fidelity_xy c1
  (opaque_trans' x c1 c2).
```

We next generalize further using general pre- and post-conditions.

Definition 3.9 (Hoare fidelity (general)). A program c_2 preserves the Hoare fidelity of a program c_1 with respect to pre-condition P and post-condition Q , if the validity of the Hoare triple $(|P|) c_1 (|Q|)$ implies the validity of the Hoare triple $(|P|) c_2 (|Q|)$. In Coq,

```
Definition Hoare_fidelity c1 c2 P Q :=
  hoare_triple P c1 Q
  -> hoare_triple P c2 Q.
```

We would like to generalize Theorem 3.8 even further and prove it for general pre-conditions P and post-conditions Q but we cannot (we show where an attempt to prove the general case fails in [10]). We will run into this exact same problem later in Section 4.3 and provide a solution there.¹⁰

3.3. A Formulation without Assignment

In the first presentation of the opaque predicate transformation from Section 3.1, we used a program that allowed the user (that is, the person obfuscating the code) to specify a particular number, and then add a number of assignments before the opaque predicate check, and then ultimately noted at the end of Section 3.1 that command equivalence (which depends on the full state — that is, the equality of values of *all* variables) did not hold in this model due to these extra assignments and variables.

We now present an alternate formulation with no assignments, with the entire predicate built into the boolean condition of the branching statement. On the one hand, the entire equation appears on a single line instead of in a number of assignments, which may make it easier to

10. AF: Section 4.3 does not mention Hoare fidelity and I don't see the connection.

detect, but on the other hand, it can access any variable already being used (and in the case of IMP, any variable at all; recall a state in IMP is a total map from strings to \mathbb{N} and all variables have default value 0). In this case, command equivalence can be proven in general.

The following two definitions are important in the theorems and examples below.

Definition

```
make_opaque_pred (a1 a2: aexp): bexp
:= BEq a1 a2.
```

Definition

```
make_opaque_pred_IFB b c1 c2
:= IFB b THEN c1 ELSE c2 FI.
```

Theorem 3.10. If a predicate b is boolean equivalent to *true*, then for any programs c_1 and c_2 , the program c_1 is command equivalent to the program resulting from applying *make_opaque_pred* to b , c_1 , and c_2 .

The power of the result stated in Theorem 3.10 is that now, the particular programs and predicate used are irrelevant and can be swapped with anything, so long as we can prove the fact that the predicate supplied is indeed an opaque predicate.

Example 3.11. We can now apply this theorem to our same running example of predicate and factorial program as before and show a general result.

```
Example example_fact_opaque_pred:
cequiv fact_nonzero
(make_opaque_pred_IFB
(make_opaque_pred
((X + 1) * (X + 1))
(X * X + X + X + 1))
fact_nonzero SKIP).
```

3.4. Weakened Information Simulation

We close this section with a series of examples that formally demonstrate the obfuscating effect of using an opaque predicate from the point of view of a simulated attacker.¹¹ We use Hoare Logic with the factorial program again (Definition 3.4), with input $X = 3$, output $Y = 6$, and with the *square_program* as the dummy program.

```
Definition square_program : com :=
Y ::= X * X.
```

Example 3.12. With the original factorial program, it is a straightforward application of the more general Example 3.5 that the Hoare triple below is valid.

$$(|X = 3|) \text{ fact_program } (|Y = 6|)$$

Next, we'll define an obfuscated program by using the transformations defined in the previous section:

```
Definition trans_fact_square_program :
com :=
(make_opaque_pred_IFB
(make_opaque_pred
((X + 1) * (X + 1))
```

11. AF: Wei suggests removing this whole subsection if we need space. We would also need to remove the reference to it at the beginning of the section.

```
(X * X + X + X + 1))
fact_program
square_program).
```

Example 3.13. With the transformed program, the analogous Hoare triple below is valid.

$$(|X = 3|) \text{ trans_fact_square_program } (|Y = 6|)$$

It is noteworthy to point out that the proof uses the known fact that our opaque predicate indeed always evaluates to true.

Example 3.14. Now to simulate an attacker who does not understand the opaque predicate, we show that the best information that can be gleaned is that the output is either 6 or 9; the proof must proceed through both the if-then and if-else branches, and the final post-condition weakened to the disjunction of the two possible outcomes, yielding the Hoare triple

$$(|X = 3|) \text{ trans_fact_program } (|Y = 6 \vee Y = 9|).$$

4. Control Flow Flattening in IMP/Coq

Today's reverse engineering tools and/or other static analysis tools can at a glance reveal some information about the *control flow* of the program, or the rough structure as delineated by the flow of blocks of code through If-Then-Else, While-Do-End, Switch, and Jump constructs.

One of the obfuscation techniques to make this difficult to analyze is *control flow flattening*. This technique aims to break apart all of the constructs that would reveal information about a program's control flow and flatten an entire program into a single semantically equivalent switch statement inside a while loop.

For a through treatment of control flow flattening obfuscation and its effects in obstructing static analysis see [13].

4.1. Flattening an If-Then-Else Construct

For the first half of this section, we will focus on a single transformation that turns an If-Then-Else construct into an equivalent flattened program (see listing 5 which is adapted from [9]):

```
int i;
i = 0; // Header
if (i < 100)
{
  i++;
}
else
{
  i = 9;
}
// Footer
```

Listing 5: If-Then-Else

```

int i;
int swVar;
swVar = 0;
while (swVar <= 4) {
  switch (swVar) {
    case 0:
      i = 0; // Header
      swVar = 1;
    case 1:
      if (i < 100) {
        swVar = 2;
        break;
      } else {
        swVar = 3;
        break;
      }
    case 2:
      i++; swVar = 4;
      break;
    case 3:
      i = 9; swVar = 4;
      break;
    case 4:
      // Footer
      swVar = 5;
      break;
  }
}

```

Listing 6: Flattened If-Then-Else

We will, in Section 4.2, first add the syntax and semantics¹² of Switch statements to the IMP language. Then in Section 4.3 we formalize the above transformation, define what it means for it to be correct, realize some additional conditions are required and formulate what those are, and then finally prove it so.

4.2. Augmenting IMP with Switch (IMP+Switch)

Before we can formalize control flow flattening of an If-Then-Else construct, we need to enrich IMP with the syntax and semantics of switch statements, which we'll call the *IMP+Switch* language. We'll also define a new type, *address*, which is just a wrapper for a *nat* and a type *lc* (list of commands) which is a list of possible switch branches indexed by *address*, and then redefine the type *com* to support switch statements.

Definition *address* := nat.

Definition *lc* := list (address * com).

Inductive *com* : Type :=

```

...
| CSwitch :
  string ->
  list (address * com) -> com.

```

Next, we create a function to search a *lc* by address.

12. Phil: Do we need the syntax and semantics? Amy: The definition of *com* was originally left out entirely from IMP without Switch in the earlier sections, so it doesn't make sense to add it here. In Section 2, I added a description and notation of *com*. This section needs to be updated to do the same.

```

Fixpoint lc_lookup
  (tlc: lc) (adr: address):
  option com := ...

```

We accordingly redefine the command evaluation semantics to include switch statements.

The following theorem and its proof already exist in [8], but we must update it for our new formulation with switch statements. The proof also introduces several new tactics and features which are worth explaining (see [10] for details).¹³

Theorem 4.1. Command evaluation is deterministic, in the sense that if a command evaluates a state *st* to a state *st1*, but also to some (possibly) other state *st2*, then it must be the case that *st1* = *st2*. In Coq,

```

Theorem ceval_deterministic:
forall c st st1 st2,
  c / st ==> st1 ->
  c / st ==> st2 ->
  st1 = st2.

```

4.3. Flattening If-Then-Else in IMP+Switch

Definition 4.2. We build an If-Then-Else statement to be flattened by the following function in Coq. The *header* and *footer* parameters are simply any IMP commands that occur before and after, respectively, the If-Then-Else statement. Since we wish to prove command equivalence between the original and transformed programs, we note that *swVar* is introduced with value 0 and ends with value 5 in the transformed program; hence, we'll preprocess the original program to be transformed by adding in these assignments.

```

Definition
  preprocess_program
  header cond c1 c2 footer:
  com := ...

```

Definition 4.3. Transforming a program with control flow flattening on an If-Then-Else statement takes the If-Then-Else components and builds a 'switch' wrapped in a 'while' with 'cases' appropriately handled.

```

Definition
  transform_program
  header cond c1 c2 footer:
  com := ...

```

We now wish state a general theorem that all programs' preprocessed and transformed forms are command equivalent. However the fact that we haven't fully accounted for the newly introduced *swVar* which controls the switch statement is problematic. If the original program already uses this variable in some way, then everything could break.

Definition 4.4 (Evaluation invariance). A program *c* is *evaluation invariant* with respect to a variable *X* if, for all states *st* and *st'* and all *n* ∈ ℕ, *c* evaluates *st* to *st'* if and only if *c* evaluates *st* updated with (*X* → *n*) to *st'* updated with (*X* → *n*).

In other words, if the only thing that changes about the start state is the value of *X*, there is no impact on

13. Phil: Are these ever explained below? Amy: No, must be modified.

evaluation with the sole exception of the same change to X in the end state. In Coq,

```
Definition eval_invariant c X :=
  forall n st st',
    c / st \\\ st' <->
    c / st & { X --> n }
    \\\ st' & { X --> n }.
```

Lemma 4.5. Evaluation invariance implies evaluation independence in the sense that, if a command c is evaluation invariant with respect to X , then if c evaluates a state st updated with $(X \rightarrow n)$ for some $n \in \mathbb{N}$ to st' , then c also evaluates st to st' . In Coq,

```
Lemma eval_inv_imp_eval_ind:
  forall c X n st st',
    eval_invariant c X ->
    c / st & { X --> n } \\\ st' ->
    c / st \\\ st'.
```

Theorem 4.6. Control flow flattening of If-Then-Else constructs is sound in the following sense.

Fix the variable $swVar$ for the control flow flattening transformation. For any program of the form
 header;; IFB cond THEN c1 ELSE c2 END
 ;; footer, we have command equivalence between the programs

```
preprocess_program ...
```

and

```
transform_program ...
```

as long as the following hold:

- The commands *footer*, $c1$, and $c2$ are evaluation invariant with respect to $swVar$.
- The boolean condition *cond* is boolean invariant with respect to $swVar$.

4.4. Flattening a While-Do-End Construct

We now switch gears and study (as well as formalize) an example of dismantling and then flattening a While-Do-End construct described in [13]

Starting with a program with a While-Do-End construct, it is *dismantled* into a number of basic blocks (sequence of non-control flow commands ending with a control flow command), essentially replacing the While-Do-End construct with conditional GoTo statements at the end of some blocks. The targets of these GoTos are determined dynamically with conditions on some variable in memory, instead of a direct (constant) address as the jump target. Following [13] we call this intermediate transformation a dismantling.

The next level of the transformation is reminiscent of the transformation studied in the first half of this chapter, wherein these GoTos are replaced by entry into a switch statement. In keeping consistent with prior terminology, this transformation will be called flattening. We'll develop a different language in the next section, however, and model this a bit differently from the switch statements of Sections 4.2 and 4.3, as we need to consider basic blocks as first class citizens.

4.5. Wrapping IMP in a Flowchart Language (IMP+Flow)

In this section,¹⁴ we describe a new lower-level formal language which will be used to represent the example from the previous section. We'll call this language *IMP+Flow*, short for flowchart¹⁵. We also define evaluation relations for basic blocks and programs of basic blocks (see [10] for details).

This language is similar to intermediate languages that are transpiled to and used in the commercial obfuscation tools such as Cloakware's obfuscation engine [14]. While very cumbersome to actually write programs in, it is well-suited to control flow related algorithms due to its treatment of basic blocks (of code) as first-class citizens.

Note here that the underlying program is the original IMP, and not the IMP+Switch defined in Section 4.2. We handle switch statements differently here, by defining them as a type of block.

4.6. Flattening While-Do-End in IMP+Flow

Starting with an example program in IMP proper:

Definition 4.7.

```
Definition OriginalCommand : com :=
  WHILE (A <= 2) DO
    B ::= A + B ;;
    IFB (! (B <= 4)) THEN
      B ::= B - 1
    ELSE SKIP FI ;;
    A ::= A + 1
  END ;;
  RETURN ::= A * B.
```

We transform¹⁶ it into the dismantled version which is basically the same program but using basic blocks and labels and jumps. We show that the 'DismantledProgram' will, like the 'OriginalProgram', evaluate a state that begins with $A = 1$ and $B = 2$ to a final state that has $RETURN = 12$. We then manually build the flattened version of 'DismantledProgram' component by component and show that the resulting 'FlattenedProgram' will, like the 'OriginalProgram' and the 'DismantledProgram' evaluate a state that begins with $A = 1$ and $B = 2$ to a final state that has $RETURN = 12$ (see [10] for details).

5. Obfuscation in Coq: Related Work

There have been three papers by Sandrine Blazy and co-authors that study code obfuscation in Coq, which we discuss here.

The first paper, *Towards a formally verified obfuscating compiler* [15] also uses IMP (their own formulation and not the one from [8]) as the language for obfuscation, but studies data and layout obfuscation techniques, as opposed to the control obfuscation techniques which

14. AF: Address Phil's comments about flowchart languages, maybe by adding a reference to the Software Foundations book, where it is used there.

15. Note that our formulation for *flowchart* is different from [8]

16. Phil: Too compressed. Maybe put the details into an appendix.

opaque predicates and control flow flattening fall under [16]. They first consider obfuscating integer constants using a function that maps each integer to a different one, and then performing a substitution using this mapping. They also consider changing variable names, which can be used, for example, to change a descriptive variable name like `account_balance` to a string of gibberish. Such techniques are inherently different from the ones we have studied. One can make a simple combinatorial argument that putting them together in the same obfuscation transformation would generate a synergistic effect, making a program possibly much more difficult to analyze.

The second paper, *Formal verification of control-flow graph flattening* [9] also studies control flow flattening, but the authors use the Clight language of CompCert [5] (the formally verified C compiler in Coq). Clight is the first intermediate language in the CompCert compiler workflow, and the strategy used was to prove the correctness of the obfuscation in that setting, from which the correctness of the compilation process follows “for free” from CompCert’s own proofs of semantic preservation.

On the one hand, the approach in [9] is less elementary and less accessible, as it works with a nontrivial subset of the real C language, but on the other it is clear evidence that formal verification of obfuscation techniques need not be restricted to a small language like IMP (which would never be used in real software development). Other real-world practicalities considered in [9] include simulation techniques and analysis of running time.

The work in [9] also discusses some techniques for combining obfuscation techniques, such as splitting a switching variable into two different variables that are updated at different points of a program, as well as randomly encoding the values of the switch cases so that they are not just consecutive numbers beginning with 1. These are necessary considerations, since we need to think one level higher about attackers, and obfuscate the fact that we are obfuscating particular parts of our code with control-flow graph flattening in the first place!

In comparing the work in [9] to ours, we believe there is merit both in the IMP and the CompCert routes. In the former, the language used is of minimal complexity, which allows not only for specifications and proofs of transformations to be developed quicker without being bogged down in unnecessarily complicated features of the underlying language, but is also better suited for pedagogical purposes (see our research goal 2). IMP is also Turing complete, so from a theoretical point of view there is no loss of generality in proofs made using it—they can always be adapted to CompCert later. But on the other hand, CompCert is, of course, closer to languages that would be of interest to real-world software development and so more practical in that sense.

The authors of [9] also ran into a similar issue as we did; they needed to separate switching variables from those in the program to be transformed, but their solution was different. They instead use a function to parse the program to be transformed and generate a fresh variable which doesn’t appear there to be used for the transformation. From a practical point of view, this approach is perhaps more natural, and in line with how a real obfuscating tool would function—generating new variables rather than demand that a certain specifically named variable doesn’t

exist in the source program. Theoretically, though, these are equivalent, since any program can contain only finitely many variable names, and there are an infinite number to choose from.

The third paper, *Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions* [17] continues to work in Clight, and studies obfuscations that involve mixing arithmetic operators and bitwise boolean operators. This is another data obfuscation that appears frequently in real-world binary code, but as it is based on features wildly beyond the capabilities of IMP, a detailed discussion is beyond the scope of the present work.

6. Future Work

The work done to date on formal verification of obfuscation, both in the present work and in the papers of Blazy et al., while providing a solid proof-of-concept that obfuscation tools of the future could support formal verification, are still limited in scope in the sense that they treat individual transformations.

A real world obfuscator mixes many different transformations together at once, often in non-deterministic ways for *diversification* of obfuscations, and so some form of compositionality would need to be implemented on these formal proofs to be able to use them together and preserve the desired formulation of correctness.

Furthermore, we (along with the work of Blazy et al.) have, in the formal setting of Coq, only tackled one desired property of obfuscation—correctness. That is, some form of the semantics of the program, or relationship between inputs and outputs, should be preserved (obfuscation property 1 mentioned in the introduction, i.e. the functionality property). But there are, of course, other properties that have not been touched upon, namely properties 2 and 3.

In closing, we stress, once more, that it *is* important to actually apply formal specifications and methods to security goals and metrics in some form, so we can come full circle and give prospective clients of an obfuscation tool a clear answer to the *other* big question “How exactly will using this improve the security of my programs?” and be able to back our answer with a proof that it actually does so.

References

- [1] R. W. Floyd, “Assigning meanings to programs,” in *Program Verification*. Springer, 1993, pp. 65–81.
- [2] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [3] B. Meyer, “Applying design by contract,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.
- [4] G. C. Necula, “Proof-carrying code,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 106–119.
- [5] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.
- [6] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, “Code obfuscation against symbolic execution attacks,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 189–200.

- [7] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy *et al.*, “The coq proof assistant reference manual: Version 6.1,” 1997.
- [8] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hricu, V. Sjöberg, A. Tolmach, and B. Yorgey, “Programming language foundations, volume 2 of software foundations,” 2010.
- [9] S. Blazy and A. Trieu, “Formal verification of control-flow graph flattening,” in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, 2016, pp. 176–187.
- [10] W. Lu, “Formally verified code obfuscation in the coq proof assistant,” Ph.D. dissertation, Université d’Ottawa/University of Ottawa, 2019.
- [11] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.
- [12] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, “Opaque predicates detection by abstract interpretation,” in *International Conference on Algebraic Methodology and Software Technology*. Springer, 2006, pp. 81–95.
- [13] C. Wang, J. Hill, J. Knight, and J. Davidson, “Software tamper resistance: Obstructing static analysis of programs,” Technical Report CS-2000-12, University of Virginia, 12 2000, Tech. Rep., 2000.
- [14] C. Liem, Y. X. Gu, and H. Johnson, “A compiler-based infrastructure for software-protection,” in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, 2008, pp. 33–44.
- [15] S. Blazy and R. Giacobazzi, “Towards a formally verified obfuscating compiler,” 2012.
- [16] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” 1997.
- [17] S. Blazy and R. Hutin, “Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions,” in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2019, pp. 196–208.