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 is 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] 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) of Necula [6].

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 code transformations such as obfuscating transformations 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,[4] 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.

Total equivalency 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

1. AF: citation needed?
2. AF: citation needed
3. AF: citation needed
4. AF: citation needed

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.

```
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

In general, though, most transformations, whether optimizations or obfuscations, but especially obfuscations which are the focus of this paper, 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 [15], obfuscators are programs that transform an input programs (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 nonlinear opaque predicate transformation 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 [12] 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 [9], 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 [2]. 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 learning this infrastructure seemed prohibitive, and thus much less accessible. 3) We want the framework to be extendable. Following the lead of [9], 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, including complete state equivalence[5] as well as Hoare logic equivalence.[6] In this particular setting, the latter is a novel approach,[7] and we give examples of its use with *opaque predicate transformations*.[8] In addition, the use of Hoare logic in this context leads to establishing our main approach to "certifying obfuscating transformations": our obfuscated programs will be "decorated" à la Pierce [9] with additional assertions whose proofs will also be provided.
- We give motivations and top-level explanations for the proofs[9] (and refer to [14] 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.
- We begin with a minimal version of IMP and augment it as needed for *control flow flattening algorithms*,[10] 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.

5. AF: It is not clear what "state equivalence" means. Do you mean the notion of equivalence already described? If so, then I think that "complete state equivalence" could be replaced by "program equivalence". If not, then explanation is needed.
6. AF: Is "Hoare logic equivalence" something we define, or a known concept? If the former, is it explained clearly later? I think we either need to say what it is here, or give a forward reference saying we will define it later.
7. AF: Again, is this novel because it is something new that we are defining or because we are using a known concept in a novel way? Clarification is needed.
8. AF: I put this in italics because it is not something we want to define here, but is defined later. We probably want a forward reference to the section where it is defined.
9. AF: I don't think this paper gives any explanations of proofs. This is a contribution of Wei's thesis, but probably not of this paper. I think it can just be removed. But somewhere, we should refer to Wei's thesis, and maybe some of this can be said at that point.
10. AF: Like "opaque predicate transformation", I put this in italics and I think it needs a forward reference.

```
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

2. The IMP language

We now give the necessary definitions and theorems from Software Foundations [9].[11]

2.1. Command equivalence

For two commands (IMP programs) c_1 and c_2 to be 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,

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

2.2. 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 2.1 (Hoare triple). A *Hoare triple*, which we sometimes refer to simply as a triple, consists 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 formulation in Coq we will need to take states into account.

Definition 2.2 (Assertion). An *assertion* about a program's state, formally, 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".

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 (see [9]).

3. Opaque predicates in IMP/Coq

An *opaque predicate* [10] is a predicate[12] that always evaluates to either *true* or *false* and the truth-value of which is known to the transformation but hard to deduce by an attacker [11]. The code under the *false* branch is never evaluated at runtime so *opaque predicates* incur no runtime performance penalty.

Of course, the absolutely 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 [11].

An *opaque predicate* transformation takes as inputs a program to be obfuscated, c_1, an opaque predicate P, and a dummy program[13] c_2, and returns the program

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

In the Section 3.1, we describe our initial (straightforward, naive) attempt, explicitly defining the transformation to introduce the lines of code that assign variables associated with the opaque predicate (as one may naturally

11. AF: As Wei mentioned, we don't define IMP, but I think we need to in order for the reader to understand the rest of the paper. We need to define `com` and `state` along with all of the notation we use for them (e.g., so that the reader can understand any of the programs like fact_nonzero). We need to define the evaluation and the notation used for it (e.g., c / st backslash st'). As well as any other basic definitions and notations used in what is now Section 3.

12. This could be any statement in a program that could evaluate to true or false, but we will only be concerned with arithmetic formulas here.

13. It's not known to an attacker, a priori, that it's a dummy program. In practice, c_2 should be constructed so that it is not obvious; e.g. c_2 should not be simply an empty program, but should be complicated enough that it looks like it could feasibly be intended to be executed.

expect to write code in a typical imperative language), and see that trying to state a general theorem about command equivalence ends up being problematic.

However, we then discuss how this spawned two ideas in different directions that rectify the issue. On the one hand, we use Hoare logic with this first formulation, in Section 3.2, to prove weaker conditions of a transformation than total command equivalence. On the other hand, in Section 3.3 we reformulate the transformation to rely on values already existing in the state of the program, with the view that one may be applying an opaque predicate transformation to a small piece of code somewhere within a much larger program that would have such values floating around in the state already.

Finally in Section 3.4, we again employ Hoare logic to give a formal example of how an attacker who does not know about the opaque predicate's constant truth valuation, but otherwise can analyze (using static analysis) the program, ends up gaining weaker knowledge because of it.

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

Example 3.2. The $fact\_nonzero$ program with input $X = 3$ yields output $Y = 6$. However, the story is not quite so simple (it is true that input $X = 3$ yields $Y = 6$, but as one can see in the Coq example, the state keeps track of the value of every variable involved in the program.). The specification of this statement in Coq is

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

Note that formally, the final state holds the information of every intermediate assignment made by the program. We can discern the output $Y = 6$ by the fact that this is the rightmost case of a value being assigned to Y. But wait, there's more! We said earlier that in Coq, an example is no different from a proposition or a theorem in anything but name, so we must actually give a proof[14]. Moreover, since command evaluation is relational and not functional (recall the reason for this is the possibility of non-terminating While loops), we must build the proof out step by step (see [7] for proofs).

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.
```

That is, the $opaque_trans$ function takes 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 is true for all x, 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.

The observant logically inclined reader should, at this point, now be suspicious about taking this claim at face value. What do we mean when we say the transformed program should be "the same"? The next example, which shows what happens when $opaque\_trans$ is applied to the $factorial\_3$ example, elucidates the necessity to be precise. First, however, we will need a few lemmas that show our opaque predicate is indeed such, in various incarnations to be used in proofs.

Lemma 3.3. It is indeed the case that

$$\forall x \in \mathbb{N}, (x * x + x + x + 1) = (x + 1) * (x + 1).$$

Example 3.4. For any $x \in \mathbb{N}$ and any program c_2, $opaque\_trans$ x $fact\_nonzero$ $c2$ 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);
```

14. This really is an example, to us. But just because one declares "here is an example of X" does not mean that X is necessarily true. In Coq, a proof must still be constructed. For example, in natural language, one can say "An example of a prime number is 20051". But this isn't immediately obvious, and one still needs to prove that example, for instance, by writing a program that tries to divide it by every number up to its square root.

```
    Z --> 3; Y --> 1; Y --> 3;
    Z --> 2; Y --> 6; Z --> 1 }.
```

After instantiating variables and unfolding the definitions, the transformed program looks like:

```
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
```

We cannot use *cequiv* (2.1) — that is, we can't use it with the current formulation of the transformation) — since new variables and assignments are introduced and kept track of in the definition of the state, even if we ultimately don't care about them.

Thus we were not ultimately successful, in this initial approach, in formulating a statement with command equivalence (2.1). We'll revisit this 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 (in our case, just the value of a single variable before the program runs and the value of a single variable after the program finishes, rather than the entire state as in the previous section). First, we'll use a slightly different formulation of the factorial program.

Definition 3.5 (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 restating Example 3.2, replacing the specific values of 3 and 6 with arbitrary natural numbers, as the Hoare triple

$$(|X = x_0|)\ fact\_program\ (|Y = x_0!|).$$

We give definitions of the assertions on 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).
```

Example 3.6. We are now ready to prove

$$(|X = x_0|)\ fact\_program\ (|Y = x_0!|)$$

which in Coq is:

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

We have proven that when $X = x_0$ before the (un-obfuscated) `fact_program` runs, then $Y = x_0!$ after the fact. We now turn to showing that when we obfuscate `fact_program`, it remains the case that $X = x_0$ beforehand implies $Y = x_0!$ when the program finishes.

In the following, we use a new formulation of the opaque predicate transformation, as it now makes our life easier to 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 3.7. We now prove the same Hoare triple holds with the obfuscated factorial program in place of the original program.

$$\forall x_0 \in \mathbb{N},\ \forall c_2 \in Com, (|X = x_0|)$$
$$opaque\_trans'\ (X, fact\_program, c_2)\ (|Y = x_0!|)$$

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) }}.
```

We've now successfully shown that our factorial program, both with and without the opaque predicate transformation, satisfies a Hoare triple of the form

$$(|X = x_0|)\ c\ (|Y = y_0|),$$

and we would like to generalize[15]. Let's introduce a new term: *Hoare fidelity*.

Definition 3.8 (Hoare fidelity (with respect to input X and output Y)). A program c_2 preserves the Hoare 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

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

15. Our result is still rather specific; 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. The pre- and post- conditions in Hoare logic could be more general, such as assertions that a variable isn't equal to some value, is greater than some value, or a conjunction or disjunction of several other statements.

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. See [14] for proof of the theorem 3.9.

Theorem 3.9. For all programs c_1 and c_2, and all $x \in \mathbb{N}$, the transformed program $opaque\_trans'(x, c_1, c_2)$ preserves the Hoare fidelity with respect to input X and output Y of c_1. In Coq,

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

Let's attempt to generalize this further[16] now.

Definition 3.10 (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 the theorem 3.9 even further and prove it for general pre-conditions P and post-conditions Q but we cannot. We will run into this exact same problem later in Section 4.3 and provide a solution there.

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 a number of assignments, which may make it easier to detect, but on the other hand, it can access any variable already being used (and in the case of IMP, also any variable not already being used; recall a state in IMP is a total map from strings to $\mathbb{N}$ and all variables have default value 0). In this case, state equivalence can be proven in general.

```
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 3.11. If a predicate b is boolean equivalent to true, then for any programs $c1$ and $c2$, the program $c1$ is boolean equivalent to the program resulting from applying make_opaque_pred to b, $c1$, and $c2$.

The power of proving a theorem to this level of generality 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 3.12. We can now apply this theorem to our same running example of predicate and factorial program as before.

```
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. Hoare logic - weakened information simulation

We close this chapter with a series of examples that formally demonstrates the obfuscating effect of using an opaque predicate from the point of view of a simulated attacker. We use Hoare Logic with the factorial program again, with input $X = 3$, output $Y = 6$, and with the $square\_program$ as the dummy program.

Example 3.13. With the original factorial program, it is a straightforward application of the more general theorem already proven that the Hoare triple

$$(|X = 3|)\ fact\_program\ (|Y = 6|)$$

is valid.

Example 3.14. With the transformed program, the analogous Hoare triple

$$(|X = 3|)\ trans\_fact\_square\_program\ (|Y = 6|)$$

is valid.

Example 3.15. Now to simulate an attacker[17] who does not understand the opaque predicate, we show that the best information that can be gleamed is that the output is either 6 or 9; the proof must proceed through

16. We've generalized to the point that the specific program no longer matters, but we are still requiring a specific subset of possible pre- and post-conditions.

17. This research is primarily focused on correctness rather than security models. We're just assuming that we have an attacker performing static analysis on the code, and that he or she doesn't know the opaque predicate is always true or always false. Under that assumption, we show that they obtain weaker information than they otherwise would have.

both the if-then and if-else branch, and the final post-condition weakened to the disjunction of the two possible outcomes, yielding the Hoare triple

$$(|X = 3|)\ trans\_fact\_program\ (|Y = 6 \vee Y = 9|).$$

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.

On of the obfuscation techniques to make this difficult to analyze, then, is *control flow flattening*, which aims to break apart all of these 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 chapter, we will focus in on a single transformation that turns an If-Then-Else construct into an equivalent flattened program (see listing 5 which is adapted from [2]):

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

Listing 5: If-Then-Else

We will, in Section 4.2, first add the syntax and semantics 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.
```

```
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

```
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.

```
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 [9], 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 [14] for details).

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 (a priori, 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 out 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 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 := ...
```

Definition 4.4 (WorldEater program). We'll use a minimal example program for this section, which we call *WorldEater*, a program that does nothing if the variable X is zero, and assigns $X = 1$ otherwise.

```
Definition WorldEater : com :=
  IFB (X = 0) THEN
    SKIP
  ELSE
    X ::= 1
  FI.
```

Example 4.5. To preprocess $WorldEater$, we feed its components to `preprocess_program`.

```
Definition PreprocessWorldEater :=
  preprocess_program
    SKIP (X = 0) SKIP (X ::= 1) SKIP.
```

Note that since the header and footer are mandatory, we add $SKIP$s, and the $swVar$ is set to the same initial and final values so we can prove command equivalence to the transformed program.

Example 4.6. To transform $WorldEater$, we feed its components to $transform\_program$.

```
Definition TransWorldEater :=
  transform_program
    SKIP (X = 0) SKIP (X ::= 1) SKIP.
```

Example 4.7. The preprocessed and transformed $WorldEater$ programs are command equivalent. In Coq,

```
Example WorldEaterTransEquiv :
  cequiv
    PreprocessWorldEater
    TransWorldEater.
```

The proof (see [14] for details) follows the same structure and ideas as the more general Theorem 4.10 to come.

We now wish to generalize this example, and 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 4.8 (Evaluation invariance). A program c is *evaluation invariant* with respect to a variable X if, for all states st and st' and all $n \in \mathbb{N}$, c evaluates st to st' if and only if c evaluates st updated with $(X \longrightarrow n)$ to st' updated with $(X \longrightarrow n)$.
In other words, if the only thing that changes about the start state is the value of X, there is no impact on 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 4.9. 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 $(X \longrightarrow n)$ 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 4.10. 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, 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 flowchart[18]. We also define evaluation relations for basic blocks and programs of basic blocks (see [14] 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 [17]. 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 4.11.

```
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 transform 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 [14] for details).

18. Note that our formulation for *flowchart* is different from [9]

5. Related works

There have been three papers, in all of which Sandrine Blazy (Université de Rennes 1) appears as a coauthor, that study code obfuscation in Coq.

Towards a formally verified obfuscating compiler

Towards a formally verified obfuscating compiler [1] also uses IMP (their own formulation and not the one from [9]) as the language for obfuscation, but studies data and layout obfuscation techniques, as opposed to the control obfuscation techniques which opaque predicates and control flow flattening fall under [8].

The first particular transformation studied herein is obfuscating integer constants, wherein all integer values are substituted by different ones in a distorted semantics using an obfuscating function $O : \mathbb{N} \to \mathbb{N}$. The other discussed is variable encoding, which changes the names of variables. A real-life application of this could be, for instance, to change a descriptive variable name like $account\_balance$ to a string of gibberish.

This is an inherently different class of techniques from the ones studied in the present work, and one can make a simple combinatorial argument that putting them together in the same obfuscation transformation would generate a synergistic effect on making a program more difficult to analyze.

Formal verification of control-flow graph flattening

Formal verification of control-flow graph flattening [2] also studies control flow flattening, but the authors use the Clight language of CompCert [4](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 strictly there, from which CompCert's own proofs of semantic preservation give the correctness of the rest of the compilation process "for free".

On the one hand, this makes the work 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), and other real-world practicalities considered in this paper include simulation techniques and analysis of running time.

This work 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 CFG flattening in the first place!

In comparing this work 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 ran into a similar issue as in the present work of needing 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 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.

Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions

Formal verification of a program obfuscation based on mixed boolean-arithmetic expressions [3] mixed boolean-arithmetic continues to work in Clight, which studies obfuscations that involve mixing arithmetic operators and bitwise boolean operators. This is another data obfuscation which 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 Dr. 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 three aforementioned pieces of related work) 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). 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] Sandrine Blazy and Roberto Giacobazzi. Towards a formally verified obfuscating compiler. In 2nd ACM SIGPLAN Software security and protection workshop, SPP 2012, St. Petersburg, FL, USA, 2012. HAL (https://hal.archives-ouvertes.fr/).

[2] Sandrine Blazy and Alix Trieu. Formal verification of controlfow graph flattening. In Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016, pages 176-187, New York, NY, USA, 2016. ACM.

[3] Sandrine Blazy and Remi 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, CPP 2019, Cascais, Portugal, January 14-15, 2019, pages 196-208, 2019.

[4] Xavier Leroy, "Formal verification of a realistic compiler," Commun. ACM 52(7): 107-115 (2009)

[5] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[6] George C. Necula: Proof-Carrying Code. POPL 1997: 106-119.

[7] Weiyun Lu, "GitHub repository to accompany the present research," https://github.com/weiyunlu/coq-certified-obfuscation, 2019

[8] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[9] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. "Programming Language Foundations,". Software Foundations series, volume 2. Electronic textbook, May 2018.

[10] Christian S. Collberg, Jasvir Nagra: Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Software Security Series, Addison-Wesley 2010, ISBN 978-0-321-54925-9, pp. I-XXVII, 1-748

[11] Mila Dalla Preda, Matias Madou, Koen De Bosschere, Roberto Giacobazzi: Opaque Predicates Detection by Abstract Interpretation. AMAST 2006: 81-95

[12] "The Coq Proof Assistant", https://coq.inria.fr, 2019

[13] Chenxi Wang and Jonathan Hill and John Knight and Jack Davidson, "Software Tamper Resistance: Obstructing Static Analysis of Programs," 2000

[14] Weiyun Lu, "Formally Verified Code Obfuscation in the Coq Proof Assistant," Collection: Theses, 2011 -, UOttawa, 2019

[15] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, Alexander Pretschner: "Code obfuscation against symbolic execution attacks," ACSAC 2016: 189-200

[16] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, Ke Yang: "On the (Im)possibility of Obfuscating Programs," CRYPTO 2001: 1-18

[17] Clifford Liem, Yuan Xiang Gu, Harold Johnson: "A compiler-based infrastructure for software-protection,". PLAS 2008: 33-44