

Towards Certified Program Obfuscation

Weiyun Lu

*School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
WLU058@uottawa.ca*

Bahman Sistany

*Cloakware Research
Irdeto Canada
Ottawa, Canada
bahman.sistany@irdeto.com*

Amy Felty

*School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
afelty@uottawa.ca*

Philip Scott

*School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
philip.scott@uottawa.ca*

Abstract—We expect our systems including software systems to function “correctly”. By “correctly”, we mean that a system will behave according to explicit and/or implicit expectations. Typically, extensive testing is done to increase the confidence in correct functionality of a piece of software but positive test results are not a proof of correctness. In *High Assurance* systems, formal verification based methods are used. Behaviour of interest such as functionality, safety and security may be expressed via some sort of formal specification language.

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? Will the transformed code preserve the specified behaviour as one expects?

To achieve the highest levels of assurance that the transformations have maintained correctness of the code, is to prove the two versions of the program (before and after the transformation) is equivalent. Total equivalency between the two versions of a program certainly implies the correctness of any specified properties of interest but what if we directly try to show validity of these properties on the transformed program?

In this thesis, we lay the foundation to study and reason about code obfuscating transformations and show how the preservation of certain behaviours may be “certified”. To this end, we apply techniques of formal specification and verification, by using the Coq Proof Assistant and IMP (a simple imperative language within it), to formulate what it means for a program’s semantics to be preserved by an obfuscating transformation, and give formal machine-checked proofs that these properties hold.

We describe our work on opaque predicates, a simple control flow obfuscation, and elements of control flow flattening transformation, a more complex control flow obfuscation. Along the way, we employ Hoare logic as our foundational specification language, as well as augment the IMP language with Switch statements. We also define a lower-level flowchart language to wrap around IMP for modelling certain flattening transformations, treating blocks of codes as objects in their own right.

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

1. Introduction

1.1. Background and Motivation

We expect our systems including software systems to function “correctly”. By “correctly”, we mean that a system will behave according to explicit and/or implicit expectations or said another way to its written and/or unwritten specifications. Typically, extensive testing is done to increase the confidence in correct functionality of a piece of software but alas testing is known to be based on inductive reasoning where more tests passing can only increase the likelihood of correctness, so positive testing results are not a proof of correctness. In systems where more assurance of correctness is required various types of deductive reasoning is used. These are formal verification methods based on theoretical foundations rooted in logic. It is important to note that, formal verification transfers the problem of confidence in program correctness to the problem of confidence in specification correctness, so it is not a silver bullet however since specifications are often smaller and less complex to express, we have successfully reduced the trusted computing base (TCB) and increased our chances of achieving correctness.

Formal verification based methods, used to show a (software) system behaves as its specification says, typically employ a specification language based on the familiar “assertions”. A specification is typically expressed in some variation of first order logic and the verification system will deductively try to prove the assertions correct or signal that they don’t hold. This is a rather elaborate process where assertions (general propositional statements about program fragments that are expected to hold) are used to generate verification conditions (VC), logic formulas, that are then fed into a satisfiability modulo theories (SMT) solver, either behind the scenes in a verification backend or in more visible to the verification expert. VC generation for program verification goes back to at least

Hoare’s triples, Eiffel style contracts and proof-carrying-code (PCC) of Necula [9].

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)?

To achieve the highest levels of assurance that the transformations have maintained correctness of the code, is to prove the two versions of the program (before and after the transformation) is equivalent. Proving equivalency is doable in certain cases but in general is still extremely hard to do for general programs. Despite the difficulty, there are now formally verified compilers such as CompCert. However, the problem with verifying realistic systems such as a compiler is scale. CompCert verification of semantic equivalence between C and generated assembly took several man years to complete. The cost of using formal verification for mere mortals (on realistic systems) is still extremely high.

Total equivalency between the two versions of a program certainly implies the correctness of any program properties of interest but what if we directly try to show validity of these properties? We will limit our solution to proving properties of interest in the “before” version of a program are maintained in the “after” version of the program (after a transformation is applied).

Certain simple transformations may be easy to discharge as they don’t invalidate expressed properties about the “before” version versus the “after” version. Below the program snippet in listing [1] asserts that $y > 2$ which we can verify visually to be true. In the snippet in listing [2] we use a simple obfuscating transformation called variable splitting where we have split the variable x into two other variables $x1$ and $x2$ and we see that (visually) 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 specially obfuscations, invalidate any assertions that hold true about the “before” version. Obfuscation is especially troublesome because the goal of obfuscation is to hide the functionality of the code from prying eyes while maintaining the functionality of the “before” program. “Prying eyes” could as easily be the same as 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 has been applied to hide the fact that at program’s end, value of z is in fact 30. We can see that the transformation makes it a bit harder to see that the assertion still holds but knowing the fact that $\forall x \in \mathbb{Z}, ((x^2 + x) \bmod 2) == 0$, we can deduce that the assertion does hold and the value of z is in fact still 30.

```
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

1.2. Summary and outline of the rest of the paper

2. 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 as the language for obfuscation, but studies data obfuscation techniques, as opposed to the control obfuscation techniques which opaque predicates and control flow flattening fall under [10].

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 $O : \mathbb{N} \rightarrow \mathbb{N}$. 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 (the formally verified C compiler in Coq, discussed in Chapter Clight is the first intermediate language in the CompCert compiler, 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 our 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, the present author believes 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. 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.

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 control-flow 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] K. Elissa, “Title of paper if known,” unpublished.
- [7] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
- [8] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetism Japan, p. 301, 1982].
- [9] George C. Necula: Proof-Carrying Code. POPL 1997: 106–119.
- [10] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.