Securing Complex Software Systems Using Formal Verification and Specification

Xuanrui Qi
xuanrui.qi@tufts.edu
Department of Computer Science
Tufts University

Supervisor: Ming Y. Chow

December 13, 2017

Abstract

Complex software systems — including low level software systems such as compilers and operating systems, as well as higher level but security-critical systems such as voting machines, blockchain implementations and network protocols — contain a myriad levels of abstraction and indirection. Therefore, these systems tend to contain numerous bugs and security vulnerabilities. Moreover, since such systems are rather complex, it is inefficient and difficult to perform high-coverage testing, and static analysis tools are usually ineffective and produce too many false positives on such large code bases.

In light of the unrealiability of static and dynamic analysis in assuring large software systems, formal verification is a rather promising and generally effective approach to secure such systems in a reliable and rigorous way. Nevertheless, there is no "silver bullet", and various caveats limit the practicality and cost-effectiveness of formal verification as a software security technique. In this paper, we will investigate a number of approaches for using formal methods to verify programs and ensure the absence of exploitable bugs, demonstrate the formalization and verification of certain properties for a short segment of code as an example of using formal verification for bug-elimination, as well as present a case study of a verified software system: the CompCert verified C compiler.

1 Introduction

In the contemporary world, *everything* runs on software, including many critical missions: for example, aircraft control software, which the safety of the life of passengers depend on, and voting machines, which can control the politics of countries. However, it is also known that almost every piece of software that is complex enough contain bugs and security vulnerabilities. For example, a piece of aircraft control software written in C might contain a good number of bugs such as null pointer dereferences and floating point precision bugs, and security vulnerabilities such as buffer overruns. It is thus important that software engineers designing and implementing such systems detect and patch as many bugs and vulnerabilities as possible, even before the software ships. Moreover, sometimes the implementation of software could be outright *incorrect* without resulting in any bugs, and such incorrect components might pass on results to other components of the software, resulting in the software exhibiting unexpected and wrong behavior; therefore, it must also be made sure that all software shipped for critical tasks are actually *correct*. In this paper, we will mainly concern ourselves with the *security* of complex software systems; that is, our goal would be to rid software of security vulnerabilities and incorrect segments which could be used to deliberately generate incorrect results.

One may think, "why don't we write a large number of tests and run them before the software is shipped?" Testing and dynamic analysis is certainly a valid and useful technique for quality and security assurance, but there are certain limitations to it. The most important concern is that testing is actually a very expensive operation: not only that there must be a team devoted to testing, but also that testing requires the tested software to be actually run, thereby requiring a huge amount of time and computational resources. Moreover, many dynamic analysis tools like valgrind slow down program execution significantly, further complicating the time and CPU resource requirements.

A more experienced software developer may think, "we could use static analysis! It's fast and it doesn't require code to be actually executed." While static program analysis is certainly also an important and effective technique to, it also has its own caveats: most interesting properties of programs are simply undecidable. Therefore, a static analysis tool must make the choice of generating potentially large numbers of false alarms or simply allow some vulnerable code to be undetected. Neither case is desirable in the development of large software systems which can potentially contain millions of lines of code.

However, there is a third alternative that receive little attention in industry, that is, formal verification. By formally *verifying* programs, we may mathematically prove certain interesting properties of programs: for example, we may be interested in proving that a merge sort program always return a non-decreasing permutation of the input list, or that a compiler always preserves the semantics of the input program. Most of these proofs will be nontrivial, but many desirable and interesting properties could be proven with a high degree of certainty. While in a Turing-complete language, there are always certain properties that are not verifiable, a programmer could usually verify most interesting properties of a program given that the program is known to terminate. Since most useful programs need to be terminating, this usually doesn't pose an important difficulty for programmers.

2 To the community

Software vulnerabilities are a difficult problem for software developers, security professionals and researchers alike. One may sort to good coding practices—essentially, programmers reasoning informally about their programs to make sure that no security exploit could possibly succeed—to secure programs. However, when programs become complex enough, it becomes much more difficult for programmers to reason informally about their programs: a C programmer may look at a 20-line segment of code and assert that no buffer overflows are possible, but the programmer could not practically inspect a 10000-line program and assert that it is impossible to attack the program by overrunning the buffer. Thus, it is important for software developers and security practitioners to know that formal verification exists—as an effective tool to help programmers reason about their programs *formally* and to aid programmers' reasoning by computer. Many security vulnerabilities—such as buffer overflows—often stem from lack of careful reasoning about code. With formal verification, one can eliminate much of the vulnerabilities of this type.

3 Major approaches to formal verification

There are many different approaches to the formal verification of programs. However, most of these techniques could be classified under three classes: model checking, axiomatic semantics, and theorem proving [9].

3.1 Model checking

Model checking is an automata-based approach to program verification which involves stating a program specification in terms of state transitions and verifying the program by checking the state space of the program. For example, we might model check an autonomous vehicle's control program only starts the engine after the door is locked; to check for this, we will, usually by using an automated tool, extract a state transition graph out of the control software, and check the aforementioned specification by checking all possible state transitions after the door is locked.

Although model checking can be highly automated, and there are already many different frameworks for model checking, it also has many deficiencies as a formal verification method. First of all, many programs—especially potentially non-terminating programs—can have infinite state spaces. Moreover, not all specifications could be easily stated, or even at all stated, in terms of state transitions: for example, it is difficult to assert that a program "never accesses memory outside of allocated bounds". Thus, as automatic and simple model checking is, it is hard to use model checking for purposes other than verifying the temporal behavior of programs.

3.2 Axiomatic semantics

Axiomatic semantics involves expressing the semantics, or meaning, of a program in terms of logical statements. One of the earliest—and still among the most important—of these axiomatic semantics were the Hoare logic [5], which consists of a *precondition*, a program statement, and a *postcondition*. If a set of preconditions are satisfied, a correct program should produce an output or a program state that satisfies the postcondition. There are also multiple extensions to Hoare logic, such as Reynolds' separation logic [10], which allow program verifiers to prove more complex facts about programs.

By composing basic axioms about individual program statements, a program verifier can prove conditions about larger programs. For example, by composing assertions that each statement in a function doesn't mutate a certain variable, we may conclude that this function cannot change the value of that variable. However, axiomatic semantic methods are useful for verifying properties about the imperative behavior and side-effects of programs, it is very difficult to prove basic facts such as "this function returns the correct result" in program logics such as Hoare Logic.

3.3 Theorem proving

"Theorem proving" in terms of program verification simply refers to writing down intended properties as propositions and then proving them as one does a mathematical theorem—usually with the aid of an interactive or automatic prover. Of the three methods, theorem proving is the least automated verification method, as it requires humans to write specifications and often also proofs of those specifications. However, theorem proving is the most powerful of the three methods, as many program properties that are verifiable under the other two methods—such as output correctness or the absence of invalid memory reads and/or writes–can be verified using theorem proving.

In principle, as a collorary of the undecidability of the halting problem, in any Turing-complete programming language, there are certain program properties which could not be verified regardless of method used; this follows from the undecidability of the halting problem. However, as we can often limit ourselves to a terminating subset of the language that is expressive enough for most purposes, this is usually not a problem in program verification.

4 A whirlwind tour of theorem proving

Theorem proving, when used for program verification, is usually done with the aid of either an interactive proof assistant such as Coq or Isabelle, or an automatic proving tool such as Z3. In addition, some programming languages, such as F*, Idris and Liquid Haskell, have type systems that are expressive enough for programs to be written and verified in the same language, and then efficiently compiled and executed. However, programs in most programming languages such as C must be verified with the aid of an external theorem prover: the abstract syntax tree of the source program and semantics of the source language must be represented in the prover.

4.1 The Curry-Howard Isomorphism

Interactive proof assistants are simply IDEs for a proof language, which is a programming language designed to write proofs. To write machine-checkable proofs, we need to exploit the **Curry-Howard Isomorphism**, which states that every type correspond to a proposition and an expression of that type correspond to a constructive proof. Thus, to prove a statement, one could simply write out its corresponding type and construct a term of that type. To prove meaningful statements, a proof language needs to have an expressive enough type system that, for example, allows programmers to include values in types (i.e., **dependently typed**). For example, the statement "the list `xs` contains 3" could be represented by a dependent algebraic data type `Elem xs 3`, and the constructors of `Elem` should correspond to the ways that membership in a list could be proven (i.e. recursively).

4.2 Automated approach: SMT Solvers

Sometimes the proof target is simple enough that an automated reasoning tool could verify its correctness; certainly, this is often preferable. When the statement is a simple statement about arithmetic or a data structure, this is often possible. With a **satisfiability modulo theory (SMT)** solver, one could verify many simple logical statements easily. Here, we will not explain the theory or implementation of SMT solvers, but shall simply note that they are often used to prove simple statements about arrays, lists, or (a restricted set of) integer arithmetic. However, in more complex systems, the specifications are often too complex for SMT solvers to reason automatically in their entirety.

4.3 Example: verifying a simple sorting function in Coq

As an example, we will verify a simple insertion sort program in Coq. Although "sortedness" is easy for a programmer to understand intuitively, the formal specification for "sortedness" is actually fairly tricky and somewhat complex. Thus, although SMT solvers might, in principle, be able to verify sortedness automatically, the formulation can be complex enough that a manual, computer-assisted direct proof is more preferable. First, we implement a simple insertion sort in functional style:

```
Require Import List.
Require Import ZArith.
Require Import Bool.

Fixpoint insert x xs :=
  match xs with
  | nil ⇒ x :: nil
  | hd:: tl ⇒ if (x <=? hd) then x::hd::tl else hd::insert x tl
  end.

Fixpoint insertion_sort xs :=
  match xs with
  | nil ⇒ nil
  | hd:: tl ⇒ insert hd (insertion_sort tl)
  end.
```

Suppose that we want to prove that the insertion sort implemented above is correct. We may write a specification for `insertion_sort` below:

```
Inductive sorted : list nat → Prop :=
| sorted_nil : sorted nil
| sorted_1 : forall x,
    sorted (x::nil)
| sorted_inductive : forall x y tl,
    (x <= y) → sorted (y::tl) → sorted (x::y::tl).
```

This specification for sorted lists is intuitive to understand: empty and 1-element lists are trivially sorted, and a sorted list remains sorted if an element not greater than the first element of the sorted list is appended to the list. We may then write a (partial) specification for `insertion_-sort`: namely, `insertion_sort` should always produce a sorted list.

```
Theorem insertion_sort_sorted: forall xs, sorted (insertion_sort xs).
```

Finally, we may prove the correctness of the insertion sort function described above in Coq, proceeding by induction on the list. However, as this paper is not a paper on Coq tactics, we will leave the complete proof in a Coq file that readers could refer to. The main step of the proof, however, is to first prove a lemma which asserts that `insert` is correct, i.e. insertion into a sorted list produces a sorted list; this proof is also done by induction on the structure of the list. The proof is presented in the file `InsertionSort.v` along with the code. Later, if we would like to use our verified insertion sort function in a larger program, we may extract OCaml code from the Coq source. Since all proofs have been checked at the time of extraction, all proofs are simply disposed of during the extraction process. `insertion_sort.ml` is the OCaml file extracted from the function `insertion_sort`.

As one may note, the proof, at 39 lines (including 5 lines of specification), is much longer than the program itself, which is only 10 lines. Proving lstinlineinsertion_sort_sorted is also much more time-consuming than writing the program itself: writing the insertion sort took only around 5 minutes, but proving the aforementioned theorem took more than 3 hours; for a more experienced Coq user, this might take much less time, but presumably still much longer than 5 minutes. Note also that the proof of `insertion_sort_sorted` is **not** a complete specification and proof of an insertion sort: a correct sort function must also guarantee that the input list has the same elements as the output list; that is, the output list must be a permutation of the input list. A function that returns an empty list regardless of the input, for example, will fit our specification. In our Coq file, we prove this property, but we will not explain the details of the proof here. However, if the reader is interested in learning about Coq as a program verification tool, we shall refer the reader to the textbook *Software Foundations* [7].

5 Evaluating the effectiveness of formal verification

5.1 Potential problems

Formal verification tools, like any software, are potentially bug-prone. Thus, there could potentially be bugs in the verification tools used, resulting in incorrect code being verified. Although this is not a significant problem, it should teach us an important lesson: there is no silver bullet. Besides software bugs in verification tools, there are many other things that could go wrong.

First of all, the specifications might be themselves incorrectly written, rendering the verification results useless. Even if they are correctly written, there might be omissions, resulting in critical properties of the source program not being proven. Finally, bugs could still occur in unverified components, which might be incorrectly assumed to be correct.

5.2 What software is worth verifying?

As software verification is a costly process, it is agreed that security-critical software components—often systems software— are most worth verifying; in other words, software that are worth verifying are those that must be trustable even in the face of the attackers. Those software include, but are not limited to, operating system kernels, compilers, web browsers, and cryptographic libraries [4]. Among those verified software systems, the most well-known are probably the seL4 microkernel and the CompCert C compiler. Next, we will briefly study the CompCert C compiler and its merits.

5.3 Case study: the CompCert C compiler

CompCert is an optimizing C compiler that supports almost all of the standard C99 language and is verified in Coq [6]. CompCert guarantees that all compiled code preserve the semantics of the source program, as dictated by the C language specifications. However, CompCert initially had only a verified backend, and the compiler frontend (which generated intermediate representation) was not verified; after a number of bugs were found in the CompCert frontend, the frontend was eventually rewritten and verified in Coq. As of 2006, an experimental version of CompCert, which contains a fully verified compiler backend, contains a total of 4378 lines of Coq code and 1313 lines of OCaml code, as well as 36282 lines of proof [6], demonstrating that a verified compiler for a small language like C is not at all impractical.

The compiler has generally been regarded as the least bug-prone part of software systems, in that it is usually believed that compilers almost never introduce bugs by incorrectly compiling programs. However, an 2011 study pointed out that compiler bugs are actually much more common than most assume: wrong-code bugs (or bugs in which the compiler generates incorrect compiled code) have been found in GCC, Clang, and all other major open-source C compilers in fairly large numbers—29 wrong-code bugs were found in GCC, while 66 wrong-code bugs were found in the LLVM [11]. CompCert is the only C compiler in which no wrong-code bugs were found by the authors. For this reason, it is important for critical software to be compiled using a bug-free compiler.

But, as mentioned above, formal verification is never bullet-proof. In addition to the bugs found in the initially unverified frontend, there were also bugs that result from incorrect assumptions made by the hardware model used in proofs [4]. Despite that CompCert is not bug-free, it does contain far fewer bugs than other mainstream C compilers, and is thus much more trustworthy than other alternatives. For this reason, Airbus had begun adopting CompCert for compiling aircraft control software [2].

However, in terms of semantics, C is a relatively simple language. Moreover, CompCert omits most of the complex code analyses and optimizations usually performed by more sophisticated compilers like GCC, producing code that is only comparable to assembly generated by GCC undert the option -O1 [6]. Thus, it is not known if verifying a compiler for a significantly more complex language (like C++ or even Rust), or verifying a highly-optimizing compiler for C, would be actually feasible. The CompCert backend was mainly written by one person (Xavier Leroy) over a relatively modest amount of time, but verification of more complex compilers might be exponentially more difficult.

5.4 The human cost of formal verification: lessons from DARPA's HACMS program

Although formal verification is a trustable method to produce correct, safe and fast code, formal verification has a great human cost. In DARPA's HACMS project, a team of programming languages and security researchers attempted to secure a quadcopter using formal methods, while a group of penetration testing professionals attempted to attack the quadcopter's software system, given full access to the source code; after each development phase, the "Read Team", or the attackers, attempted to exploit vulnerabilities and gain control to the quadcopter without hardware access [4]. To secure the quadcopter's software system, many security properties need to be verified: for example, the software kernel used, seL4, provided verified isolation between safe and unsafe system components [8, 4], while the low-level programming language used, Ivory, had verified memory safety properties [4].

The HACMS team was able to reuse many readily-available verified software artifacts such as seL4, but many of the software used must be engineered and verified from scratch. Moreover, since availability of formal methods experts required to implement and verify software is scarce—there are probably less than 5000 of them, almost all of them whom possess PhD degrees—creating formally verified software can be extremely costly [4]. Although there are contractors which specialize in formal methods, such as Galois, Inc. and Data61, these firms are primarily staffed by researchers with PhDs and thus extremely expensive. Moreover, it is easy to write software but difficult to reason about them (perhaps this is the fundamental cause of many software bugs and vulnerabilities in the first place). Thus, a verified software artifact can take much more time to develop than its counterpart, even when taking into account that the developers of verified software are already experts; most of the extra time is spent on proofs, which can be 50-100 times longer than the code itself [4].

From the case studies of CompCert and HACMS, we see that it is perfectly practical to verify complex software systems and create verified programs that are efficient enough for production-level use; there is also an immense need for certain software systems to be verified, as there is currently no other method that is as effective as formal verification in preventing bugs and vulnerabilities. Nevertheless, the sometimes astronomic cost of verifying large software systems can impede formal verification from being used in many cases, as only very few institutions—such as the military (like DARPA), manufacturer of security-critical devices (such as Airbus), and, to a lesser extent, institutions that rely on cryptographic security—would find formal methods cost-effective.

6 Applications

Although the discussion above may generate an impression that it is impossible for formal methods to be used for most purposes, this is not true. Most developers lack the expertise and the time to fully verify the software they have created, but they may nevertheless still benefit from formal methods.

One important takeaway is that a developer could make use of verified tools instead of their unverified counterparts when the security of the software they are creating is paramount. For example, the writer of an important device driver might want to use CompCert instead of GCC to prevent potential wrong-code errors that might lead to vulnerable and/or incorrect code running in kernel mode with full system privileges. An embedded systems engineer designing an embedded system for a critical task might consider using the seL4 microkernel to ensure that no attacker could gain control of the system because of vulnerabilities in the kernel. A blockchain implementer might want to consider using a verified implementation of cryptographic functions [3] so that transactions do not go wrong because of bugs in the cryptography implementation. Software developers could eliminate many vulnerabilities in critical software by using verified tools, *even if the software is not itself verified.*

In addition, while full verification of complex software systems is difficult and costly, it is often possible to write partial proofs about a certain parts of the software, and there are multiple tutorials to doing this, such as the textbook *Software Foundations* [7]. For example, developers at IBM have prototyped a query compiler in Coq, proving some (but not all) properties of the compiler along the way [1]; this is just one possibility for incorporating formal methods into a large software developement project when end-to-end verification is not possible. Despite that non-verified components of the software might still contain bugs, this is still a considerable step forward.

7 Conclusion

In conclusion, formal verification is a powerful and trustable way to assure that software do not go wrong or contain vulnerabilities. Formal verification methods could be used to prove important properties about programs—from memory safety to cryptographic correctness— at the time of development, and allows developers to still ship high-quality, performant code. However, although formal verification seems almost perfect (barring human errors caused by carelessness), there is a practical problem with formal verification, as formal verification requires a high level of expertise and a lot of developer time. Thus, to make greater use of formal verification as a security method, we must train more developers well-versed in the basics of formal methods—including but not limited to basic programming languages theory, basic cybersecurity knowledge, and some familiarity with interactive proof assistants like Coq.

References

[1] Auerbach, J. S., Hirzel, M., Mandel, L., Shinnar, A., and Siméon, J. Prototyping a Query Compiler Using Coq (Experience Report). *Proc. ACM Program. Lang. 1*, ICFP (Aug. 2017), 9:1–9:15.

[2] Bedin França, R., Favre-Felix, D., Leroy, X., Pantel, M., and Souyris, J. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *PPES 2011: Predictability and Performance in Embedded Systems* (Grenoble, France, Mar. 2011), vol. 18 of *OpenAccess Series in Informatics*, Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, pp. 59–68.

[3] Beringer, L., Petcher, A., Ye, K. Q., and Appel, A. W. Verified correctness and security of openssl HMAC. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 207–221.

[4] Fisher, K., Launchbury, J., and Richards, R. The HACMS program: using formal methods to eliminate exploitable bugs. *Phil. Trans. R. Soc. A 375*, 2104 (2017), 20150401.

[5] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

[6] Leroy, X. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2006), POPL '06, ACM, pp. 42–54.

[7] Pierce, B. C., Azevedo de Amorim, A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. *Software Foundations.* Electronic textbook, 2017. Version 5.0. `http://www.cis.upenn.edu/~bcpierce/sf`.

[8] Potts, D., Bourquin, R., Andresen, L., Andronick, J., Klein, G., and Heiser, G. Mathematically Verified Software Kernels: Raising the Bar for High Assurance Implementations. Technical report, NICTA, Sydney, Australia, July 2014.

[9] Ray, S. *Scalable Techniques for Formal Verification.* Springer US, 2010.

[10] Reynolds, J. C. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002), pp. 55–74.

[11] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not. 46*, 6 (June 2011), 283–294.