Software Security by Certified Software

Shen Liu
Department of Computer Science and Engineering
Lehigh University
19 Memorial Drive West
Bethlehem, PA, 18015
shl413@lehigh.edu

***Abstract*—Software security is one of the most pressing challenges confronting individuals and organizations. Traditional methods for improving the software system security, such as software testing, cannot provide 100% trustworthiness. For critical software with higher requirements on security, using formal machine-checkable proofs can promote the software trustworthiness to a great extent. This paper briefly introduces the concept of certified software. We point out its necessity, describe its basic framework, and summarize the recent advances in this area. We use a simple example to show the basic formal proof process, and a more practical example to demonstrate how to verify a realistic program. At last, we summarize the research and propose several potential topics for our own research in future.**

***Keywords*-certified software; formal proof; verification; Coq;**

I. Introduction

Software security is a very challenging problem for all software developers. For a long time, developers have used testing to find insidious bugs and improve the quality of their products. However, the developers still cannot assure that the software is free of bugs only by testing. Because computer software is becoming more and more complex and even the developers themselves cannot know precisely how the software systems work in practice. That means software testing cannot cover all states during the program execution [1], and a program may crash when it runs into a state which has not been tested.

Computer scientists have tried to promote the software security for a long time. However, they often encounter common difficulties. For example, in this field, using quantitative techniques will be quite difficult, because metrics are still lacking for measuring software security. So, a software with five bugs is not necessarily five times more secure than a software with only one bug. In some worse cases, a software often depends on its execution environment, such as hardware, user operation and the physical world. So, it is very difficult to set up benchmarks to compare different techniques and build steady progress in this field. On the other hand, for application software, the security depends on their underlying system software (e.g. OS kernel and compiler) [1]. Bug-prone low-level programs are not trustworthy; the security of related application software could be compromised.

Certified software technique aims to tackle these problems. Developers can use formal specifications to state the software behaviors they desire, and verify that executable machine code can behave as expected [1]. In this case, formal claims that the developers make can be considered as a dependably metric. For difficulties caused by bug-prone system software, the developers need to certify the corresponding system software code. By doing this they can provide a more secure system stack for related application software.

In this paper, we survey this topic, introduce the basic idea of certified software and discuss challenging open problems that we will focus on in the future. The rest of the paper is organized as follows: Section 2 gives basic concepts and typical framework about certified software; Section 3 uses a simple verification example to show the main process of verifying a section of C code; Section 4 presents a more practical example—-the formal verification of a realistic compiler. Section 5 states the practicality and benefit of using formal proof rather than testing, and points out that using software verification is more cost-effective than traditional testing methods; the conclusions and perspectives are given in Section 5.

II. Concepts and Framework

A. What is Certified Software?

Shao summarizes the basic idea of certified software framework in [1]. In his summary, a typical certified software consists of a machine-executable program C, a formal proof P (checkable by computer) and a set of dependability claims S. Both the proof P and the specification S are written using a mathematical logic (also a programming language), and can be developed using software tools (e.g. proof assistants, automated theorem provers, and certifying compilers). The proof P can be checked automatically for correctness on a computer, by a small computer program (checkable by hand) called proof checker. If the logic used by programmers is completely consistent, and the dependability specification accurately describes what the users want, then programmers can believe that the software is free of bugs with respect to the specification.

B. Framework

As shown in Figure 1, a typical certified framework consists of four components as follows:

Machine code and formal proof. This part is the certified software itself;

Dependability claim. Providing preconditions and post-conditions that must be satisfied by the software (e.g. safety property, security policy, and functional specification for correctness);

Underlying mechanized metalogic. Logic for writing all proofs, specifications, and machine-level programs (not shown in Figure 1);

Proof checker. A small program for checking the validity of all the proofs.

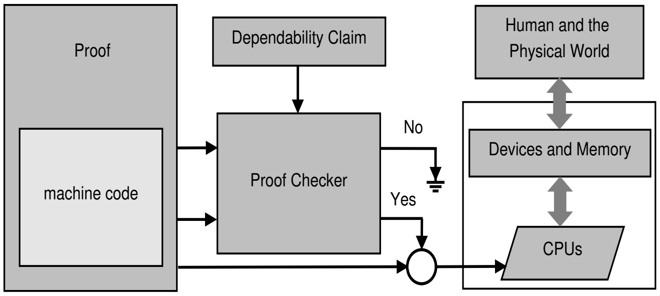

Figure 1. Components of a typical certified framework (copied from [1])

If the proof of a given certified software can be validated by the proof checker, then the execution of this software can be guaranteed to satisfy the dependability claim. On the other hand, considering that the proof checker itself is a computer program which could be bug-prone, the designers use simple logic with a small number of inference rules to write it, and verify it by hand, then make it the proof checker. By doing this the trustworthiness of the proof checker can be assured absolutely.

For certified software itself, writing formal proofs and specifications and checking the validity of them by computer automatically is a core technique. According to Curry-Howard correspondence [3], in programming language theory and proof theory, types corresponds to propositions, programs corresponds to proofs and computation corresponds to simplification. If we prove the codes, the correctness will be ensured absolutely. Currently, this work is finished using the proof assistant [4], which provides a rich higher-order logic with powerful inductive definitions. By using Coq, the framework could support explicit machine-checkable proof objects, and simplify the validation process by proof checker.

III. A simple Verification Example

The most challenging part of certified software framework is how to write formal proof for a section of code using proof assistant. Simply speaking, what we need to do on verification must satisfy two properties. On one hand, the code should be free of probable threats, such as dead loop, arithmetic overflow and out-of-bounds array access; on the other hand, the code must meet the given functional requirements. Both the two properties can be written as annotations, which can be inserted directly into the source code like C code comments. A simple real example is shown in Figure 2.

```
/*@ requires y/2 <= x <= 2*y
  @ ensures  \ result == x-y
  @*/

float Sterbenz(float x,float y){
  return x-y;
}
```

Figure 2. Sterbenz—-a simple C source code sample

The simple single precision C function in Figure 2 formally states the well-known Sterbenz theorem [5]: if *x* and *y* are floating point numbers not too far away ($y/2 \leq x \leq 2y$), then $x - y$ is computed exactly. In this example, the preconditions and post-conditions of this function are inserted as comments of shape /*@...*/. The keyword `requires` introduces all preconditions, before the function is carried out, these required properties must be satisfied first; the keyword `ensures` introduces all post-conditions, when the function returns, these properties must be assured; the keyword `\result` describes the return value.

When run on the source code with such annotations, the formal verification tool at the C source code level will produce a single verification condition in Figure 3 automatically.

```
∀x : single. ∀y : single.

s_to_r(y)/2 ≤s_to_r(x) ≤ 2×s_to_r(y) ⇒

    s_to_r(sub_single(nearest_even, x, y)) =

    s_to_r(x)−s_to_r(y)
```

Figure 3. Intermediate verification for Sterbenz function

The meanings of related keywords in the above intermediate code are as follows:

`single`: projection to data type, here means single precision float

`s_to_r`: projection to access the floating-number part;

`sub_single`: projection to subtraction on the type `single`;

`nearest_even`: projection to the rounding mode, by default, rounding to nearest, ties to even.

Next, this verification condition can be transformed using Coq tactics, here is the core part of corresponding proof by

Sylvie Boldo [8]:

```
...
Proof.
  intros x; intros;
  apply bounded_real_no_overflow_single.
  cut (0 <= single_value y)%R; try intros.
  cut (0 <= single_value x)%R; try intros.
  case (Rle_or_lt(single_value x)(single_value y));
  intros. rewrite Rabs_left1; ring_simplify.
  apply Rle_trans with(-0+single_value y)%R;
  auto with real. ring_simplify;
  rewrite <- (Rabs_right (single_value y));
  [apply single_le_strict|auto with real.]
  apply Rplus_le_reg_l with(single_value y);
  now ring_simplify. rewrite Rabs_right;
  ring_simplify.
  apply Rle_trans with(single_value x-0)%R.
  apply Rplus_le_compat_l; auto with real.
  ring_simplify;
  rewrite <- (Rabs_right(single_value x));
  [apply single_le_strict|auto with real].
  apply Rle_ge;
  apply Rplus_le_reg_l with(single_value y);
  ring_simplify; auto with real.
  apply Rle_trans with(single_value y/2)%R;
  [idtac|apply HW_1].unfold Rdiv;
  apply Rmult_le_pos; auto with real.
  apply Rmult_le_reg_l with 3%R;
  [apply Rlt_le_trans with 2%R;auto with real|idtac].
  apply Rplus_le_reg_l with(single_value y);
  apply Rmult_le_reg_l with(/2)%R; auto with real.
  replace (/2*(single_value y + 3*0))%R
  with (single_value y/2)%R by(unfold Rdiv; ring).
  apply Rle_trans with(single_value x);
  [eapply HW_1|idtac].
  apply Rle_trans with(2*single_value y)%R;
  [eapply HW_1|right;field].
Save.
```

Figure 4. A section of proof in Coq

The keywords such as `intros`, `apply` and `rewrite` stand for Coq tactics, which are used for constructing complex proofs semi-automatically. The keywords like `bounded_real_no_overflow_single` and `Rplus_le_reg_l` are lemmas defined by provers. This section of Coq proof code seems quite long, and the whole proof online is even much longer [9]. However, most of code will be generated by Coq assistant automatically, and the real proof process is not very difficult. The proof written by Coq will be checked for validation by proof checker in next step.

IV. A REALISTIC EXAMPLE—CERTIFIED COMPILER

In recent years, the verification technique has been used in various types of practical software projects, such as assembly programming languages, compilers and proof assistants. Necula's and Lee's work on proof-carring code (PCC) is the immediate precursor to the large body of more recent work on certified software [12]. Yu et al. in [13] propose a low-level language, Certified Assembly Programming, for building certified programs and present a certified library for dynamic storage allocation. McCreight et al. in [14] develop a general framework for certifying a range of garbage collectors. Feng et al. in [15] certify the partial correctness of a preemptive thread library extracted from our simplified OS kernel.

Leroy verifies a realistic compiler for the first time [2]. Compilers are software that perform complicated semantic and symbolic transformations, and also bug-prone low-level programs. That means if the compiler itself is not trustworthy, it may generate incorrect executable machine code probably, no matter how correct the source program is. Figure 2 shows a simple common bug in the GCC compiler [10]. Here a bug appears because C99 allows intermediate results to be computed with excess precision, but requires them to be rounded at assignments.

```
#include <stdio.h>

void test(double x, double y)
{
  const double y2 = x + 1.0; //not rounded to 64 bits
  if (y != y2) printf("error");
}

void main()
{
  const double x = .012;
  const double y = x + 1.0; //rounded to 64 bits

  test(x, y);
}
```

Figure 5. A bug caused by GCC

For non-critical software such as web game applications, compiler-caused bugs are often negligible because the developers rarely run into them. But when they have to tackle this, debugging will be very difficult. For critical software like military and financial software systems, this insidious threat should be seriously considered. The traditional testing methods are no longer effective here, and need to be replaced by using formal methods such as program verification. Otherwise, the test plan will be very complicated and expensive. However, even the guarantees obtained by formal verification in source code level may not lead to 100% trustworthy executable code because of the possible miscompilation, which means the compiler creates incorrect output from correct input.

A solution to this tough problem is using certified compiler. The core idea of certified compiler is using formal proof to the compiler itself to prove that the compiler preserves the semantics of the source code. If the compilation succeeds, the generated machine code should behave as prescribed by the semantics of the source code. Leroy et al. develop and verify a realistic compiler–CompCert, which is useable for critical embedded software. The source language of CompCert is named Clight. Clight is a very large subset of C programming language. It supports most of data types in C, such as `struct`, arrays and pointers; it also supports all C-style expressions (including pointer arithmetic), control (loops, `break`, `continue` and `if/then`), functions (including recursive functions and function pointers) and volatile accesses. The target code can be PowerPC, ARM or x86 assembly language code [7].

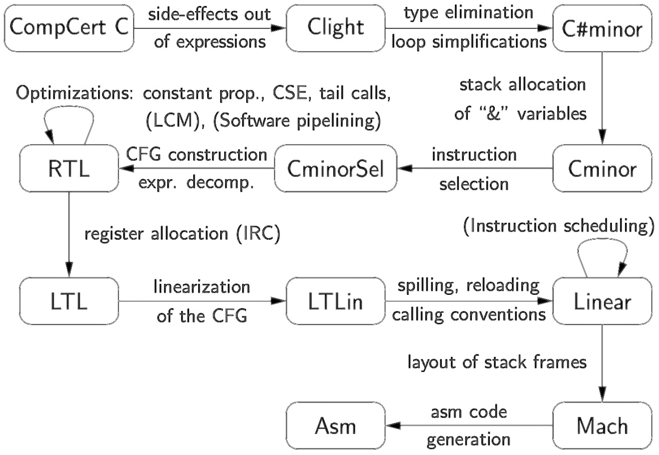
CompCert C
side-effects out of expressions
Clight
type elimination loop simplifications
C#minor
stack allocation of "&" variables
Cminor
instruction selection
CminorSel
CFG construction expr. decomp.
RTL
Optimizations: constant prop., CSE, tail calls, (LCM), (Software pipelining)
register allocation (IRC)
LTL
linearization of the CFG
LTLin
spilling, reloading calling conventions
Linear
(Instruction scheduling)
layout of stack frames
Mach
asm code generation
Asm

Figure 6. The formally verified part of the compiler (copied from [2])

The whole compliation process for CompCert is shown in figure 6. All the compilation steps in CompCert are standard steps used in all compilers, but each step is proved correct using Coq assistant, and the correctness proofs are the special part which is different from other compilers.

In the beginning, the compiler translates source Clight code into C#minor and Cminor intermediate code, during these two steps, all type-dependent behaviors will be removed. Then, the compiler selects suitable arithmetic instructions, such as add-immediate and not-and, to rewrite Cminor code to CminorSel, which is a processor-dependent variant of Cminor. Additional operators and addressing modes will be introduced in this step. Next, the CminorSel will be translated to register transfer language (RTL). The developers do optimizations in this step, because in RTL control can be represented as a control-flow graph and each node corresponds to a machine-level instruction. When optimizations are finished, the compiler will allocate registers to generate a language called location transfer language (LTL). In this subprocess, the compiler use registers or stack locations to replace previous temporaries.

Now, the compiler will carry out code linearization to linearize the control-flow graph, and replace LTL using lists of instructions with explicit labels and branches to these labels in the style of assembly code. The branches include both conditional and unconditional ones. The output target language is LTLin. Usually, two or three transitions in LTLin correspond to one intra-function transition in the original LTL code. The next step finishes the register allocation process, using ”spilling” and ”reloading” methods [2]. Instructions like `move` are inserted around function calls in this step, too. After this, the compiler will allocate space for stack and turn the accesses to stack into memory instructions `load` and `store`. The output is called Mach, which is very close to the final assembly language in semantics. In the final compilation step, the real assembly language code will be generated.

The original semantics are proven to be preserved by Coq in each step. The author's experiments show that the final executable machine code generated by CompCert outperforms the one generated by GCC without optimizations, and even competitive with GCC at optimization level 1 (`gcc -O1`) and 2 (`gcc -O2`). In 2011, Yang et al. build CSmith, which is tool for finding bugs in C compilers, and use it to perform random testing among C compilers like GCC and Clang [16]. Until now they have found 79 bugs in GCC, however, they found no bug in CompCert.

V. Testing or proof

As can be seen from above, using formal proof to improve the software security is a very complicated and effort-consuming process. A probable question is whether this technique should be adopted by industry. Because small software companies may not afford such a cost financially. However, on the other hand, the technique people are using in reality nowadays, software testing, is labor-intensive, too. The core question is for those non-critical software development, to improve the security, which one is more cost-effective?

Steve King et al. answer this question, and show both the advantages and feasibility of industrial-scale proof on commercial software projects, and show that using proof is more cost-effective than traditional testing method in [11]. They use formal proof and testing at the same time to find the faults in a real commercial software called SHOLIS project, which consists of 133,000 lines of Ada code. The authors compare the proof with various types of testing (e.g. unit testing, integration testing and system validation testing), and make analysis based on engineering judgment instead of theoretical computer science. The final result shows that the overall performance of industrial-scale proof is much better than the authors expect originally. Z proof, which is a formal proof technique they use in the paper, even outperforms the most efficient testing method on finding faults.

VI. Summary and future work

Certified software is a fast developing technique. In this paper we simply review some recent classical papers on this topic. Among all the components in a certified software framework, how to write formal proof for source code is the most confusing part for beginners. We use simplest example to illustrate the basic working process of formal verification, using both a simple C program example and a practical certified complier example. We also take the SHOLIS project for example to illustrate that using verification for quality assurance is more cost-effective than testing.

After the CompCert experiment, the goal of proving the correctness of a realistic compiler appears feasible. Considering the fact that the methodologies, techniques and tools used are far from perfect, Leroy's work opens up many directions for our own work in future. For example, we can extend Leroy's work to handle a larger subset of C, deploy and prove the correctness of more optimizations. Further, we can set up a model to prove the correctness of an assembler and linker, at least constitutes a first step towards these more complex goals, using formal methods and verification tools.

Considering the complexity of CompCert, adding new features (e.g. `goto` sentence) or optimization to CompCert may cause existing programs to fail. Because new bugs might be introduced into the proved compiler, and we have to prove the whole compiler again to check its trustworthiness. So, in the near future, we will focus on performing translation validation [17] with formal proof methods, which requires only the verification of the compiler output, not the whole compiler implementation itself.

The principal challenge on this topic is expressing algorithms as provable Coq functions. We will use inductive predicates with inference rules to specify an algorithm in our framework, and by doing this we can make the most of previous work like [18]. Our semantic functions will be constructed as follows: First of all, we state the correctness of a semantic function f as a property relating the input and its output. At the same time, we state specification Q for the function f as its preconditions and postconditions. Secondly, we create a function called *certifying function* to return two things: the justified result of the output of f (denoted as R), and a proof that shows R satisfies Q. Next, when the function f calls other functions, the related certifying functions and proofs will be given accordingly. Finally, we can combine all subproofs using appropriate logic rules to demonstrate the whole translation is proven to be validated.

This method will be very hard for long and complex functions probably, but we will keep trying. We hope to establish a more effective translation validator framework for CompCert with Coq proof assistant, which is helpful to the researchers in this area.

Acknowledgment

We would like to thank Prof. Daniel Lopresti, Prof. Gang Tan, and Dr. Maryann DiEdwardo for their helpful and valuable feedback on earlier versions of the paper.

References

[1] Z. Shao, Certified software. *Communications of the ACM*, 53(12):56-66, 2010.

[2] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107-115, 2009.

[3] C. Haskell, Functionality in Combinatory Logic. *In Proceedings of the National Academy of Sciences 20*, pp. 584-590, 1934.

[4] G. Huet, C. Paulin-Mohring, et al. *The Coq proof assistant reference manual*. The Coq release v6.3.1, May 2000.

[5] P. H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.

[6] S. Boldo, J.C. Filliâtre. Formal verification of floating-point programs. *In IEEE Symposium on Computer Arithmetic*, pp. 187-194, 2007.

[7] The CompCert verified compiler. http://compcert.inria.fr/doc/

[8] Sylvie Boldo's homepage. https://www.lri.fr/~sboldo/research.html

[9] Full Coq proof of Sterbenz's theorem. https://www.lri.fr/~sboldo/progs/Sterbenz_why.v

[10] Bug 323–optimized code gives strange floating point results. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=323

[11] S. King, J. Hammond, R. Chapman, and A. Pryor. "Is Proof More Cost-Effective Than Testing?" *IEEE Transanctions on Software Engineering*, 26(8):675-686, 2000.

[12] G. Necula, P. Lee. Safe kernel extensions without run-time checking. *In Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, pp. 229-243, 1996.

[13] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *In Proceedings of the 2003 European Symposium on Programming*, vol. 2618 of LNCS, pp. 363-379, 2003.

[14] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. *In Proceedings of 2007 ACM Conference on Programming Language Design and Implementation*, pp. 468-479, 2007.

[15] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. *In Proceedings of 2008 ACM Conference on Programming Language Design and Implementation*, pp. 170-182, 2008.

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. *In 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 283-294, 2011.

[17] A. Pnueli, M. Siegel, and E. Singerman, Translation validation. *In Tools and Algorithms for Construction and Analysis of Systems*, vol. 1384 of LNCS, pp. 151-166, 1998.

[18] J. B. Tristan, X. Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. *In 35th Symposium Principles of Programming Languages*, pp. 17-27, 2008.