Certified Binaries for Software Components

Sagar Chaki, James Ivers, Kurt Wallnau
Software Engineering Institute

Introduction

There is an evident need for mechanisms that enhance our ability to *trust* third-party software. In the current era of plug-and-play, off-the-shelf programs are being increasingly made available as modules or *components* that can be attached to an existing infrastructure. More often than not, such plug-ins are distributed in machine code or *binary* form. In this article we present a framework that can be used to generate trustworthy binaries for software components, and to prove that binaries generated elsewhere satisfy specific policies. At the core of our methodology lies a paradigm called proof-carrying-code (PCC), originally proposed in a seminal paper by Necula and Lee [1, 2]. The essential idea underlying PCC is to construct a proof of the claim that a piece of machine code respects a desired *policy*. The proof is shipped along with the code so that it may be independently verified before the code is deployed.

To date, the application of PCC has been restricted to pure *safety* policies. The progress of PCC has also been hindered by, among other things, the need for manual intervention (e.g., discovering complicated loop invariants), and large proof sizes. Our approach overcomes these limitations of PCC in the context of certifying software components using powerful, but specific techniques. In particular, we achieve the following three objectives: (1) **Enrich:** Expand the set of PCC policies to include both safety and *liveness*. To this end, we use a state/event-based temporal logic called SE-LTL developed in the context of the Predictable Assembly from Certifiable Components (PACC)[1] project at the SEI. (2) **Automate:** Use iterative refinement in combination with predicate abstraction and model checking to generate appropriate invariants and ranking functions required for certificate and proof construction in a completely *automated* manner. (3) **Compact:** Use state-of-the-art Boolean satisfiability (SAT) technology to generate extremely small proofs. Preliminary investigations [5] indicate that the use of SAT yields proofs sizes that are several orders of magnitude more compact than when using conventional methods.

Background

In the original formulation of PCC, the world is divided into trusted code consumers and untrusted code producers. A code consumer publishes a safety policy. In general, safety policies assert that “something bad never happens,” while liveness policies assert “something good will eventually happen.” The code producer annotates the code with key invariants and uses a *certifying compiler* to generate object code as well as a verification condition (VC); in essence, the VC is the logical formula that is valid if and only if the object code respects the safety policy. The certifying compiler also constructs a proof of the VC, which is embedded in the object code; hence “proof carrying.” The code consumer checks that the proof is valid by verifying its construction against a set of sound axioms and inference rules that have been defined on the machine instructions themselves. The verification step is efficient, and reduces to a form of type checking. In other words, the proof is valid if, and only if, it is well-typed.

[1] http://www.sei.cmu.edu/pacc

PCC does not depend on the correctness of the certifying compiler or on the technologies used to construct proofs of program properties. PCC is also resilient to tampering, including code optimizations. Most attempts at modifying either the object code or the proof of the VC will lead to an ill-typed proof and hence will be detected. Moreover, any undetected tampering is guaranteed to result in code that still respects the published safety policy, and hence is harmless as far as the policy is concerned. Last, proof-carrying code is efficient, since the static proof eliminates the need for runtime checks. Still, a number of technical challenges (discussed in [2]) arose in this original formulation of PCC. Particularly notable among these are:

- **(CH1)** The restriction to safety conditions is problematic if the cost of developing a trustworthy PCC infrastructure is great.
- **(CH2)** The proof generator sometimes requires manual assistance, for example to compute loop invariants; for practical transition purposes, this is a non-starter.
- **(CH3)** The proofs generated are often quite large, hindering wider use of the PCC paradigm. Despite a lot of recent advances, this problem continues to be open.

Prior to this work, we conducted two projects that have a direct bearing on the above challenges. First, as part of the PACC project, we developed an expressive linear temporal logic called SE-LTL that can be used to express both safety and liveness claims of component-based software. In this work, we adopt and modify SE-LTL to express certifiable policies, thereby targeting **CH1**. Second, in collaboration with Prof. Peter Lee, an original proponent of and leading expert in PCC, we conducted an Independent Research and Development (IRAD) project [3] on "Assessing and Demonstrating the Readiness of Proof Carrying Code for Obtaining Objective Trust in Software Components". As part of this PCC-IRAD, we have developed an infrastructure to generate compact certificates for *C programs* (not binaries) against SE-LTL claims in an automated manner. The automation is achieved by combining iterative refinement with predicate abstraction and model checking to generate appropriate invariants and ranking functions that are required for certificate and proof construction. The tightness of proofs is obtained via the use of the state-of-the-art Boolean satisfiability (SAT) technology [5]. In this work, we extend this framework to certify *binaries* generated from component specifications. Thus, we complete the framework from a PCC perspective, and also address issues **CH2** and **CH3**. To this end we build on the PACC infrastructure for analyzing specifications of software component assemblies and generating deployable machine code for such assemblies.

Overall Approach

Our technical approach is best summarized by the architecture described in Figure 1. This figure depicts the final infrastructure for certified component binary generation that we developed. The boxes are numbered for ease of reference. The steps involved in generating certified component binaries can be summarized as follows:

1. We begin (box 1) with a specification of a component assembly written in the Construction and Composition Language (CCL) [9], which has been developed as part of the PACC project. A CCL specification contains a description of the assembly as well as safety and liveness policies that need to be certified. CCL is currently implemented as a profile of an executable subset of UML 2.0.
2. The CCL specification is automatically *interpreted* [4] into a form that can be processed by a model checker. This form (box 2) essentially comprises of a C

program along with finite state machine specifications for library routines invoked by the program. The interpretation procedure was implemented as part of the PACC project.

3. The result of the interpretation is input to *Copper* (box 3), a state-of-the-art *certifying software model checker*. Copper [8] was originally developed as part of the ComFoRT [7] reasoning framework of the PACC project. It was enhanced [5] with the ability to generate certificates and proofs as part of the PCC-IRAD [3]. Copper interfaces with theorem provers (TP) and SAT solvers (SAT) during model checking and certificate generation. The output of Copper is either a counterexample to the policy (CE) or a proof (Proof1) that the input to Copper respects the desired policies.
4. Proof1 only certifies that the result of interpreting the original CCL specification respects the desired policies. It is *reverse-interpreted* (arrow 4) into a proof (Proof2) that the CCL specification itself also respects these policies. However, in order to generate certified binaries, we perform two additional steps.
5. The CCL specification is now transformed (arrow 5) into a Pin/C program (box 6) that can be compiled and deployed in a Pin runtime environment (RTE). This transformation process, as well as the Pin RTE, has been developed to a large extent as part of the PACC project. We enhanced this transformation process so that it also creates a proof of the correctness of the generated Pin/C code from the proof of the correctness of the CCL specification. In essence, we transform the proof of correctness, along with the actual assembly, from one format (CCL) to another (Pin/C).
6. The final step (arrow 7) is conceptually the same as the previous step. We use a standard C compiler (gcc) to achieve this goal. The end result is a proof (Proof3) that the final (i.e., binary code for the) component assembly respects the desired policies.

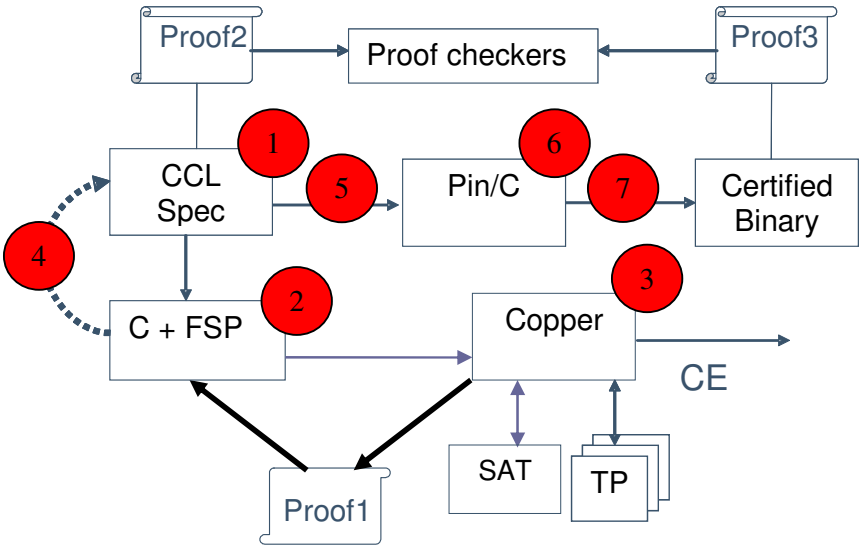

Figure 1: Architecture of developed framework.

References

[1] George Necula, "Proof-carrying code," In Proc. 24th ACM Symposium on Principles of Programming Languages (POPL), New York, Jan. 1997 (106-119).

[2] George Necula and Peter Lee, "Safe kernel extensions without runtime checking," In Proc. 2nd USENIX Symposium on Operating System Design and Implementation (OSDI), Seattle, Wshington1996 (229-243).

[3] Sagar Chaki, Kurt Wallnau, "Proof-Carrying Code," CMU/SEI-2005-TR-020, Chapter 6, December 2005.

[4] James Ivers, Nishant Sinha and Kurt Wallnau, "A Basis for Composition Language CL", (CMU/SEI-2002-TN-026).

[5] Sagar Chaki, "SAT-based Software Certification", in Proc. Of TACAS, 2006.

[6] Fred Schneider, "Enforceable Security Properties," in ACM Transactions on Information and System Security, Vol. 3, No. 1, February 2000 (30-50).

[7] James Ivers and Natasha Sharygina, "Overview of ComFoRT: A Model Checking Reasoning Framework", (CMU/SEI-2004-TN-018).

[8] Sagar Chaki, James Ivers, Natasha Sharygina and Kurt Wallnau, "The ComFoRT Reasoning Framework," in Proc. 17th Computer Aided Verification (CAV), LNCS 3576 (164-169), July 2005.

[9] Kurt Wallnau and James Ivers, "Snapshot of CCL: A Language for Predictable Assembly", (CMU/SEI-2003-TN-025).