Certified Memory Management for Proof-Carrying Code: A Region-Based Type System and Runtime Library (Extended Abstract)

Nadeem Abdul Hamid
Berry College
Mt Berry, GA 30149-5014
nadeem@acm.org

Categories and Subject Descriptors
F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms
Security, Languages

1. PROBLEM AND MOTIVATION
Proof-Carrying Code (PCC) is a generic framework intended to facilitate safe execution of code from an unknown or untrusted source. The basic idea of PCC, as the name implies, is that a piece of executable code comes packaged with a proof of its safety according to some user-specified policy. The development of this technology has been fueled by research over the past decade in the use of type systems and logic to verify properties of low-level code (*i.e.* assembly or binary machine code).

To date, the majority of such research has focused only on carrying through the compilation of source code from a single high-level language to verifiable object code. There has not been significant progress in combining such compilation with certification of that code which is only written at the systems programming or assembly code level. The runtime systems of existing frameworks thus include numerous components that are not addressed in the PCC safety proof [1, 2]: low-level memory management libraries, garbage collection, debuggers, marshallers, *etc.* Furthermore, the issue of producing a safety proof for code that is compiled and linked together from two different source languages has not been addressed, to a great extent.

In this work, we outline a PCC framework that allows for the construction of certified machine code packages from typed assembly language, based on a high-level type system. The compiled code from a high-level type system will interface with a similarly certified, low-level memory management runtime library. Details of some portions of this framework have been previously published [4, 3, 10].

43rd ACM Southeast Conference, March 18-20, 2005, Kennesaw, GA, USA. Copyright 2004 ACM 1-58113-000-0/00/0004 ... $5.00.

2. BACKGROUND AND RELATED WORK
The concept of PCC was introduced by Necula and Lee [6, 5] and a number of variants have been produced by others. In particular, an alternate approach to PCC, based on foundational principles of mathematics and logic was first pursued by Appel [1] and others. This family of PCC implementations, known as Foundational Proof-Carrying Code (FPCC), aims at an increased assurance of safety by minimizing the amount of code in the FPCC framework that must be trusted. Our work is based on the FPCC framework and extends it as outlined in Section 1 and the following sections.

To produce a PCC package, we must express the concept of safety that we wish to enforce in some way that can be mechanically checked. This is done by using a powerful calculus based on higher-order predicate logic to encode the definition and operational semantics of the machine on which code will be run. We then define the safety policy in terms of the machine's encoding. Besides an initial machine state (*i.e.* an executable binary), we also require a code producer to provide a proof that no matter how many steps the machine executes from the initial state, the safety policy will always be satisfied.

In order to build the safety proof, we provide a specification layer on top of the raw machine encoding which allows one to use Hoare logic-like reasoning to show that a piece of code satisfies desired properties. An initial development of this layer has been previously described in [10]. A fundamental principle of PCC is that it is much easier to check a safety proof than to produce one. Using only Hoare logic to build a safety proof would be extremely difficult for a code producer, especially since the ideal goal is that the production of such proofs would be as automated as possible. Thus, most source programs will be written in, or at least compiled to, a version of typed assembly language (TAL). Type-preserving compilation from a high-level language, like Java, to a low-level intermediate language, like TAL, has been widely studied. Much PCC work has focused further on the automatic generation of a safety proof from TAL code.

3. APPROACH AND UNIQUENESS

In our case, we have used a TAL with a high-level region-based memory management type system that allows the allocation and deallocation of regions of memory. The type system is based on the capability calculus of [8]. This allows the programmer explicit, but safe, control over the creation and deletion of memory objects in a program. In this work, we have formalized the region-based TAL within the PCC logic and mechanically proved its soundness. Then, we show how to compile TAL programs automatically to a Hoare logic-based safety derivation in the specification layer described in the previous paragraph.

While most user programs can be written in or compiled to TAL, there are some pieces of code which cannot be written in TAL itself – for instance, the actual implementation of the memory management runtime system. In particular, the runtime system consists of standard malloc/free code which manages a free list and pointers to allocated regions. To certify this code as safe, we must directly produce a proof using the low-level reasoning system. The novel feature of our work is that we also have shown how to "link" the automatically-generated safety proofs of compiled code to the manually generated low-level proof of the runtime library. That means that the entire application in this system can be shown to satisfy the safety policy.

There are a number of novel features of our development. First, we have formally encoded and proven the soundness of a version of TAL with a non-trivial region-based type system. Second, we demonstrate a new approach to automatically generating a safety proof in the low-level reasoning system, based on the compilation of a well-typed TAL source program. Finally, the biggest advancement is the integration of safety proofs from different sources and levels of the code generation process. As mentioned above, descriptions of some of the pieces of this framework have been previously published. However, the latest, and ongoing development has been the instantiation of this framework with a region-based memory management library.

4. RESULTS AND CONTRIBUTIONS

The initial development of FPCC made it seem extremely difficult to produce safety proofs based only on the foundations of mathematical logic. Our research has been aimed at reducing the complexity of such a task. To this end, we introduced previously a "syntactic" approach to FPCC which shows the potential of substantial improvement with respect to the ease and scalability involved in constructing an FPCC system. The syntactic approach involves encoding a high-level source type system directly in the safety logic, and proving the type system's soundness using the standard syntactic style of proof [9].

In the current development, we have refined the syntactic approach by inserting a common Hoare-logic based specification layer, which can serve as the target for compilation from any number of higher-level type systems. This specification system then also allows for low-level reasoning about particularly complex runtime library code that could not otherwise be programmed safely in most standard type systems.

For our prototype implementation, we have used a simple, idealized machine (to avoid dealing with the technical details of a "real" machine such as the Intel x86 family). We also use the Coq proof assistant [7] for the formal development of proofs. While there are currently limitations of our prototype framework,[1] we believe that the ongoing work on engineering and technical aspects will be able to produce the deliverable of a realistic certified system for the next generation of Proof-Carrying Code.

5. REFERENCES

[1] A. W. Appel. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] K. Crary and S. Sarkar. A metalogical approach to foundational certified code. Technical Report CMU-CS-03-108, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, Jan. 2003.

[3] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proceedings 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223, pages 118–135, Sept. 2004.

[4] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof carrying-code. *Journal of Automated Reasoning (Special issue on Proof-Carrying Code)*, 31(3-4):191–229, Dec. 2003.

[5] G. C. Necula. Proof-carrying code. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, Jan. 1997.

[6] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[7] The Coq Development Team. The Coq proof assistant reference manual. Coq release v8.0, 2004.

[8] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, 2000.

[9] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[10] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004.

[1]The Coq tool, for example, has not been developed at all with the purpose of PCC in mind and thus is not best suited for a realistic implementation.