A Proof-Carrying Code Architecture for Java

Christopher Colby, Peter Lee, and George C. Necula

Cedilla Systems Incorporated
4616 Henry Street
Pittsburgh, Pennsylvania 15213
Hackers@CedillaSystems.com

1 Introduction

In earlier work, Necula and Lee developed *proof-carrying code* (PCC) [3, 5], which is a mechanism for ensuring the safe behavior of programs. In PCC, a program contains both the code and an encoding of an easy-to-check proof. The validity of the proof, which can be automatically determined by a simple proof-checking program, implies that the code, when executed, will behave safely according to a user-supplied formal definition of safe behavior. Later, Necula and Lee demonstrated the concept of a *certifying compiler* [6, 7]. Certifying compilers promise to make PCC more practical by compiling high-level source programs into optimized PCC binaries completely automatically, as opposed to depending on semi-automatic theorem-proving techniques. Taken together, PCC and certifying compilers provide a possible solution to the code safety problem, even in applications involving mobile code [4].

In this paper we describe a PCC architecture comprising two tools:

- A thin PCC layer implemented in C that protects a host system from unsafe software. The host system can be anything from a desktop computer down to a smartcard. The administrator of the host system specifies a safety policy in a variant of the Edinburgh Logical Framework (LF) [1]. This layer loads PCC binaries, which are Intel x86 object files that contain a `.lf` section providing a binary encoding of a safety proof, and checks them against the safety policy before installing the software.
- A software-development tool that produces x86 PCC binaries from Java `.class` files. It is implemented in Objective Caml [2]. From a developer's perspective, this tool works just like any other compiler, with an interface similar to `javac` or `gcc`. Behind the scenes, the tool produces x86 machine code along with a proof of type safety according to the Java typing rules.

The demonstration will use a small graphics program to show that this architecture delivers Java safety guarantees without sacrificing the performance of native compilation.

George Necula's current address is Computer Science Division, University of California, Berkeley, 783 Soda Hall, Berkeley, CA 94720.

⋆ Appeared in the Tool section of the Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00), Chicago, 15 July 2000.

2 Architecture

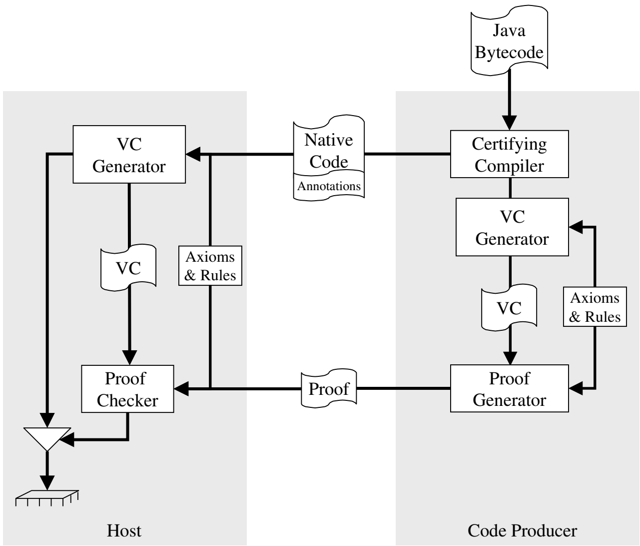

Fig. 1. The architecture of our PCC implementation.

Figure 1 shows the architecture of our PCC system. The right side of the figure shows the software-development process, and the left side shows the secure host system. VC stands for "verification condition".

The right side of Figure 1 is a code-production tool that the user runs offline to generate Intel x86 PCC binaries from Java bytecode. First, a compiler generates x86 native code from a Java `.class` file. This compiler is largely conventional except that it attaches some logical annotations to the resulting binary. These annotations are "hints" that help the rest of the system understand how the native code output corresponds to the bytecode input. These annotations are easy for the compiler to generate from the bytecode, but would be difficult for the rest of the system to "reverse engineer" from unannotated native code.

The annotated x86 binary is then analyzed by a tool called a verification-condition generator, or *VC generator*. The VC generator is parameterized by a set of axioms and rules, specified in the Edinburgh Logical Framework (LF), that describe the safety policy against which the binary must be proven. The VC generator outputs a logical predicate, also expressed in LF, that describes a precondition that, if true, would imply that any possible execution of the binary is safe. Intuitively speaking, it performs this task by scanning each native-code instruction and emitting safety conditions as they arise. The result is called the verification condition, or *VC*.

Finally, the VC is sent to an automated theorem prover, which, using the same axioms and rules, attempts to prove the VC and, if successful, outputs the resulting logical proof in binary form. The annotations and proof are added to the binary as an `.lf` segment, thus producing a PCC binary. This object file can be loaded and linked with existing tools just like any other object file, in which case the `.lf` section is ignored. Currently, proofs are 10–40% of the code size, but preliminary results with a new proof-representation technology indicate that this can be decreased to 5–10% [8].

Now we turn to the host on the left side of Figure 1. The host runs a thin PCC layer that loads PCC binaries and verifies them. The host first separates the annotated binary from the proof. In Figure 1, these are shown already separated. It then runs a VC generator on the annotated binary to produce a VC from a safety policy specified by the same set of rules and axioms.[1] Lastly, it checks the proof to make sure that it is indeed a valid proof under the safety policy. If it checks out, then the unannotated binary is installed on the host system. The annotations and the proof are discarded.

The PCC layer on the host illustrates another key engineering point that makes the PCC architecture viable—checking a proof is usually much faster and simpler than generating a proof. It is important that it is *fast* because it is happening dynamically, as software is downloaded into the host. It is important that it is *simple* because the trusted computing base (TCB) on the host must be bug-free. Furthermore, some of our current applications involve small embedded systems that lack memory resources to run large programs. For these reasons, it is unacceptable to run a complex verifier on the host system.

In contrast, PCC development tools such as the code producer on the right side of Figure 1 can be slow and buggy without compromising the soundness of the basic architecture that we are proposing in this demonstration. For instance, the compiler is free to perform aggressive optimizations, even though that might present difficult and time-consuming problems for the proof generator, because proof generation is done offline. Technically, this result is provided by a soundness theorem [6]. This result is not specific to any particular code-production tool, to Java, or to any particular safety policy, but rather is a property of the PCC architecture itself.

3 Demonstration

We have implemented prototypes of both sides of Figure 1 for the Intel x86 architecture. The certifying compiler performs register allocation and some global optimizations, including a form of partial redundancy elimination. The VC generator and proof checker are quite complete and have been stable for several

[1] The system that we demonstrate shares the same VC generator between the two sides of the architecture, and this is often a convenient approach. Conceptually, however, the host's module is part of the TCB and thus must be bug-free, while the code producer's module is not trusted, and so bugs will simply produce incorrect binaries that will be caught by the host before installation.

months. The compiler and proof generator, on the other hand, are still under heavy development. At present, the compiler handles a large subset of the Java features, including objects, exceptions, and floating-point arithmetic. However, there are several key features that have yet to be implemented, including threads and dynamic class loading. Also, a number of important optimizations are not yet finished, including the elimination of null-pointer and array-bounds checks, and the stack allocation of non-escaping objects.

The demonstration will use a small graphics program to show that this architecture delivers Java safety guarantees without sacrificing the performance of native compilation. The demonstration will compare three different approaches to transmitting untrusted code to a secure host:

1. Java bytecode, verified and run by a JVM on the host (safe but slow)
2. x86 native code produced by a C compiler, run via the Java Native Interface (JNI) on the host (fast but unsafe)
3. x86 native code produced by our certifying compiler, run via JNI on the host (fast and safe)

We demonstrate the safety of our approach (3) by showing that various forms of tampering with the PCC binary cause the host to reject the binary as potentially unsafe.

In future work, we plan to release our current system for public use. Also of great interest is to extend the safety policy to go beyond Java type safety, in particular to allow enforcement of some constraints on the use of resources such as execution time and memory.

References

1. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
2. Xavier Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, 1990.
3. George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996.
4. George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1997.
5. George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.
6. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.
7. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

8. George C. Necula and Shree P. Rahul. Oracle-based checking of untrusted software. Submitted to Programming Language Design and Implementation, PLDI'00, November 1999.