Programs from Proofs – Approach and Applications*

Daniel Wonisch, Alexander Schremmer, Heike Wehrheim

Department of Computer Science
University of Paderborn
Germany
wehrheim@upb.de

Abstract: Proof-carrying code approaches aim at the safe execution of untrusted code by having the code producer attach a safety proof to the code which the code consumer only has to validate. Depending on the type of safety property, proofs can however become quite large and their validation - though faster than their construction - still time consuming.

Programs from Proofs is a new concept for the safe execution of untrusted code. It keeps the idea of putting the time consuming part of proving on the side of the code producer, however, attaches no proofs to code anymore but instead uses the proof to *transform* the program into an *equivalent* but *more efficiently verifiable* program. Code consumers thus still do proving themselves, however, on a computationally inexpensive level only.

In case that the initial proving effort does not yield a conclusive result (e.g., due to a timeout), the very same technique of program transformation can be used to obtain a zero overhead runtime monitoring technique.

1 Overview

Proof-carrying code (PCC) as introduced by Necula [Nec97] is a technique for the safe execution of untrusted code. The general idea is that once a correctness proof has been carried out for a piece of code the proof is attached to the code, and code consumers successively only have to check the correctness of the proof. The technique is *tamperproof*, i.e., if the proof is not valid for the program and property at hand, the code consumer will actually detect it.

Within the Collaborative Research Center SFB 901 at the University of Paderborn we have developed an alternative concept for the safe execution of untrusted code called *Programs from Proofs* (PfP) [WSW13a]. Like PCC it is first of all a general concept with lots of possible instantiations, one of which we have already developed. Our concept keeps the general idea behind PCC: the potentially untrusted code producer gets the major burden in the task of ensuring safety while the consumer has to execute a time and space efficient procedure only. The approach works as follows. The code producer carries out a

*This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

proof of correctness of the program with respect to a safety property. In our current instance of the PfP framework, the safety property is given as a protocol automaton and the correctness proof is constructed by the software verification tool CPACHECKER [BK11] using a predicate analysis. The information gathered in the proof (in our scenario an abstract reachability tree) is next used to *transform* the program into an *equivalent*, more efficiently verifiable (but usually larger wrt. lines of code) program. The transformed program is delivered to the consumer who – prior to execution – is also proving correctness of the program, however, with a significantly reduced effort. The approach remains tamper-proof since the consumer is actually verifying correctness of the delivered program. Experimental results show that the proof effort can be reduced by several orders of magnitude, both with respect to time and space.

Besides using it as a proof-simplifying method, PfP can also be used as a *runtime monitoring* technique [WSW13b]. This might become necessary when (fully-automatic) verification only yields inconclusive results, e.g., because of timeouts or insufficient memory. In this case, PfP will use the inconclusive proof, i.e., the inconclusive abstract reachability tree (ART), for the program transformation. An inconclusive ART still contains error states since the verification did not succeed in proving (or refuting) the property. When transforming this ART into a program, we insert HALT statements at these potential points of failure so that the program stops before running into an error state. The obtained program is thus safe by construction, and equivalent to the old program on its non-halting paths. Thus the Program From Proofs method in this application scenario gives us a zero-overhead runtime monitoring technique which needs no additional monitoring code (except for HALTs).

References

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. In G. Gopalakrishnan and S. Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 184–190. Springer-Verlag, Berlin, 2011.

[Nec97] George C. Necula. Proof-carrying code. In *POPL 1997*, pages 106–119, New York, NY, USA, 1997. ACM.

[WSW13a] Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim. Programs from Proofs - A PCC Alternative. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 912–927. Springer, 2013.

[WSW13b] Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim. Zero Overhead Runtime Monitoring. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 2013.