Tutorial: Automated verification of systems software with Serval

Luke Nelson, Emina Torlak, and Xi Wang
University of Washington
https://unsat.cs.washington.edu/projects/serval/

Abstract

Serval is a framework for writing scalable automated verifiers. This tutorial will give a brief overview of recent progress in verifying systems software, a hands-on guide on how to formally specify and verify a system using Serval, and an in-depth discussion of formal verification techniques that power Serval.

ACM Reference Format:
Luke Nelson, Emina Torlak, and Xi Wang. 2019. Tutorial: Automated verification of systems software with Serval . In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 1 page.

1 Goals

Formal verification is an attractive approach for developing high-assurance systems. It is a timely topic in systems research, with growing interest among the systems community in applying verification to system design and implementation. This tutorial has three goals:

- presenting an overview of recent efforts on verifying systems software (with a focus on SOSP/OSDI);
- teaching how to formally specify and verify a system using automated verifiers from Serval [1], and
- illustrating the verification technologies that power Serval: the Rosette programming language [2], symbolic evaluation, and SMT solving.

2 Target audience

The target audience is researchers, students, and developers who are interested systems verification. The tutorial includes an overview of the field and hands-on experience with a verification tool, and introduces the Serval methodology for automated verification of systems software.

3 Organization

The tutorial consists of the following three parts.

(i) Overview of research on verifying systems software. The first part gives an overview of the goals and history of systems software verification. It describes recent examples of verified systems from SOSP/OSDI and the methodologies used to verify them. Furthermore, it discusses the guarantees and trade-offs afforded by the various methodologies, and gives a sense of which verification methodology may be appropriate for particular domains and properties.

(ii) Verifying systems with Serval. The next part demonstrates how to apply Serval to verify a system. It uses a toy OS kernel running on RISC-V as the example system to be verified. The audience is expected to run the automated verifiers provided by Serval on the toy OS kernel to identify bugs and to verify the fixed version. For example, the audience can use Serval's verifiers to find undefined behavior, functional correctness, and high-level safety bugs in the toy OS kernel. These examples are simplified from bugs we discovered when verifying real-world systems. For each example, this part provides skeleton code for the audience to experiment with and run on their own laptops.

(iii) Extending Serval with Rosette. The last part shows how to extend Serval using the Rosette programming language. It includes a brief tutorial on how to use Rosette for tasks such as symbolic evaluation and verification, and how Rosette reduces a problem into SMT constraints. It uses a toy instruction set as an example to show how to write a verifier for the instruction set in Rosette and how to implement symbolic optimizations to scale verification in Serval. This part also gives examples of a broader range of applications of Rosette, including verifying radiation therapy software and synthesizing problem-solving strategies for games and education.

References

[1] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP).* Huntsville, Ontario, Canada.

[2] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* Edinburgh, United Kingdom, 530–541.