Static Assertion Checking of Production Software with Angelic Verification

Shaobo He
University of Utah

Shuvendu K. Lahiri
Microsoft Research

Akash Lal
Microsoft Research

Zvonimir Rakamaric
University of Utah

Abstract

The ability to statically detect violations of assertions can add great value to developers. However, in spite of decades of progress in program verification, static assertion checking is far from being cost-effective for production software. The two main obstacles to finding high-quality defects are (a) false alarms due to under-constrained environment, and (b) finding violations to deeply nested procedures.

In this talk, we will describe our experience with the *angelic verification* (AV) tool for statically finding assertion violations in real-world software. The basic idea of AV is to pair an interprocedural assertion verifier with a framework for automatic inference of *likely specifications* on unknown values from the environment. We will summarize the approach, and will focus on design choices required to find high-quality violations of memory-safety violations with low false alarms. We discuss some results on Microsoft codebases and open source software, and challenges ahead.

ACM Reference Format:
Shaobo He, Shuvendu K. Lahiri, Akash Lal, and Zvonimir Rakamaric. 2017. Static Assertion Checking of Production Software with Angelic Verification. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Assertions provide a mechanism for specifying the absence of a large-class of runtime errors (such as absence of `null` dereference, or out-of-bounds access) and enforcing program-specific invariants. Such assertions can be either automatically instrumented or written explicitly by the developer. These assertions can either be checked at runtime or can be statically discharged. Statically finding assertion violations can provide great value to developers to find and fix bugs before deployment without the need for expensive testing or incurring the runtime overhead in production.

However, statically checking assertions on real-world programs is far from cost-effective to be deployed as part of a development cycle. First, since most static assertion checkers assume a *closed* program, a programmer has to manually constrain the environment with specifications, or create a driver (to restrict the input states) and stubs (for external methods) for any open program. In the absence of such an upfront effort, static checkers (that are often based on satisfiability modulo theories (SMT) solvers) report numerous *dumb* false alarms due to the unconstrained environment. Second, such assertion checkers do not scale to large programs when assertions are present in deeply nested procedures, since the checker starts analysis from the entry-point of the module. Note that these problems persist even when the checker sacrifices soundness by performing bounded unrolling of loops and recursion.

In this talk, we will present the *angelic verification* project [6] for addressing this issue. We summarize the approach in Section 2, and discuss some applications in practice in Section 3.

2 Angelic Verification

Angelic verification [6] (AV) is a technique for leveraging automatic static assertion checkers for finding high-confidence defects in production software. The technique pairs a precise assertion checker AC (that can find interprocedural counterexamples of assertion violation in closed programs) with the inference of *angelic specifications* on the environment, to push back on the verifier from reporting "dumb" false alarms.

Given a module M, AV starts interprocedural exploration starting from *each* procedure p in the module to find a violation. For each such check, the environment consists of the precondition of p and the summaries of any external procedure outside M being invoked from p. AV inspects a defect t reported by AC, and attempts to compute a *likely* specification over the unknowns in t that would block the defect. It reports the defect only if no such likely specification can be constructed. The ability to start analysis from any procedure in a module allows AV to scale to find defects

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

present in deeply nested procedures. The challenges in computing such specifications lie in (a) scalability of inference, and (b) avoiding overly strong assumption that may hide true defects.

AV provides several *knobs* for analysis designers to control the expressiveness of the specifications, that in turn helps determine the scalability and the shape of the specifications. For example, AV is parameterized by a *vocabulary V* of predicates that constitutes the atoms of the specifications. For example, the presence of spurious *aliasing* is often a root cause of false alarms reported by static verifiers. Hence AV allows the vocabulary to consist of aliasing prediates which can be composed using Boolean connectives. Further, AV allows the user to only suppress the *data flow* (i.e. treating the conditionals in a path as non-deterministic) or suppress the *control flow* of the defect [4, 6].

3 AV in practice

AV framework has been instantiated to create several checkers for Microsoft and open-source C/C++ codebases. We currently use Corral [8] as the underlying interprocedural assertion checker, where the programs are expressed in the Boogie language. This makes it possible to adapt the toolchain to other languages for which translations to Boogie exist. Corral unrolls loops and recursion (upto a bound) and performs an abstraction-refinement guided assertion-directed search to find violations of assertions. We have instantiated AV for two classes of generic memory-safety assertions. The first `nullcheck` checks for presence of null dereferences in a program. The second `use-after-free` checks for the presence of the use of an already freed pointer. Both these properties require deep interprocedural analysis, without well-known modular verification strategies. We also leverage the SMV [3] infrastructure to scale the massively parallel analysis using Microsoft Azure cloud. The source code for AV is hosted on Github within the Corral repository [1]. We currently have two different instances of AV that we describe in the next few paragraphs.

The first instance works with the Microsoft C/C++ `cl` compiler based infrastructure for generating Boogie [5, 7]. We have extensively evaluated this tool for `nullcheck` on several modules in Windows source code, shown parity with existing mature tool such as PREfix (details here [6]) and found several new defects. More importantly, we reported high quality interprocedural defects with very low false alarms. AV now ships with the `Static Driver Verifier` (SDV) tool for third-party driver developers and powers the `nullcheck` rule [9]. Our preliminary evaluation of `use-after-free` has also found several defects and shows promise. Much appeal for AV comes from its "pay-as-you-go" model where more modeling of the environment leads to more defects, but can provide an "out-of-the-box" value even in the absence of any modeling.

The second instance works with the `LLVM` compiler infrastructure using the SMACK [10] software verification toolchain for generating Boogie programs. We have used this version to search for memory-safety defects in several open-source C/C++ modules including `PX4` (software for autopiloting drones), Linux drivers, ssh and SQLite. Several defects discovered in Linux drivers were confirmed by the developers, to one of which a patch was immediately issued. We have also investigated defects found by AV and the state-of-the-art verifier `Infer` [2] and will report preliminary findings on relative strengths and weaknesses.

4 Conclusion

In this talk, we will present the angelic verification (AV) project, the underlying technology and some preliminary use-cases. We will outline research challenges and opportunities for addressing scalability and precision needs that will make AV more widely applicable.

References

[1] 2017. Angelic Verifier (AV). https://github.com/boogie-org/corral/tree/master/AddOns/AngelicVerifierNull. (2017).

[2] 2017. Infer tool. http://fbinfer.com/. (2017).

[3] 2017. Static Module Verifier (SMV). https://github.com/Microsoft/Static-Module-Verifier. (2017).

[4] Sam Blackshear and Shuvendu K. Lahiri. 2013. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 209–218.

[5] Jeremy Condit, Brian Hackett, Shuvendu K Lahiri, and Shaz Qadeer. 2009. Unifying type checking and property checking for low-level code. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 302–314.

[6] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, Vol. 9206. Springer, 324–342.

[7] Akash Lal and Shaz Qadeer. 2014. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 202–212.

[8] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, Vol. 7358. Springer, 427–443.

[9] Microsoft. 2017. NullCheck rule (wdm). https://msdn.microsoft.com/en-us/library/windows/hardware/mt779102(v=vs.85).aspx. (2017).

[10] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 106–113.