Automated Coverage-Driven Dynamic Testing for Assurance

Alexey Loginov *Vice President of Research, GrammaTech, Inc.* alexey@grammatech.com

1 Benefits

Testing has been a part of the Software Development Life-Cycle (SDLC) from the start. Developers validate partially constructed systems with unit tests and testers evaluate full systems with end-to-end and integration tests. In modern software organizations, much of the testing is automated in continuous integration/continuous deployment (CI/CD) pipelines.

A major challenge with traditional testing is its limited coverage: manually constructed tests are unlikely to reveal all issues, e.g., because tests created by those involved in the system's design and implementation are biased towards testing anticipated behaviors.

Testing-based (dynamic) approaches explore an under-approximation of the effectively infinite number of possible states of a system. There exist many notions of test coverage from statement coverage, to branch coverage, to Modified Condition/Decision Coverage (MC/DC) that is used for testing most critical (Level A) avionics software, following standards DO-178B and DO-178C. However, achieving such guarantees requires extensive time and expertise.

Static analysis gets around the issue of analyzing infinite-state systems by performing abstraction. By over-approximating the system's potential states, it can provide guarantees of the system's correctness. While applying a static analysis can be automatic, the design of an analysis for a new setting and the review of the resultant warnings can be very labor-intensive.

Further, static analysis of realistic systems cannot achieve perfect precision and recall (theoretically or in practice). While there exist highly tuned analyzers that aim to provide guarantees in a narrow setting (e.g., the avionics of a specific plane), most commercial static analyzers strike a balance between providing assurance and overwhelming analysts with false positives. Using such tools still requires significant configuration, a review of many warnings (to confirm true positives), and analysis of possible blind spots.

As a complement to static analysis, coverage-driven dynamic testing provides the following benefits:

- Generally, dynamic testing does not rely on source code or build scripts for the tested system. Most static analyzers operate on the source code. (Although, there exist advanced static analyzers that handle stripped machine code.)
- Dynamic testing provides broad coverage of problems–as long as an issue is detectable at runtime (e.g., as a crash or a violation of some condition), it will be identified if a triggering input is provided. Static analysis, on the other hand, requires the implementation of checks or models for each potential issue.
- Dynamic analysis produces no false positives–an error detected with an input that was accepted by the tested system should be addressed.
- Dynamic testing produces explainable results: one may guess the likely cause of problems based on just the input or can follow the flow of the input through the test system in a debugger.
- Test-case generation, e.g., using fuzzing, enables automated test-input construction. This leads to increasingly effective test suites over time.

These benefits make coverage-driven dynamic testing more informative and conclusive than static analysis alone, especially in the absence of source code. These benefits are well-known and we recommend that these techniques be used in many SDLC phases (release testing, acceptance testing, penetration testing, sustainment, and sparingly in CI/CD), as well as for more platforms (Linux, Windows, embedded devices, SCADA/ICS systems, etc.). However, there exist challenges to wider adoption of these capabilities. Below, we discuss some of the challenges and suggest promising techniques for addressing them.

2 Challenges

1. The most obvious challenge for dynamic testing is achieving thorough coverage (without putting the burden of test creation on humans). How can we generate new inputs that are driven to increase test coverage and how can we maximize the chance of detecting subtle problems that might not manifest themselves as crashes?

Position Statement for Area 4 (initial minimum requirements for testing software source code)

2. To ensure relevance to the common scenario in which a customer (or integrator) does not receive source code, we need to enable these capabilities for binary executables and libraries.
3. Another big challenge is how to run realistic applications. Traditional fuzzing, for instance, explores extensive variations on a given input (command-line or file) but isn't prepared to handle a variety of input channels, such as files, networks, GUI, etc. Can we automate the creation of harnesses, capable of supplying a variety of input types to complex applications?
4. Finally, while fuzzing has been applied extensively to Linux applications, little is available for firmware or even Windows. Can we extend the support with the same level of automation and efficacy to all platforms?

3 Promising Solutions

We have been working to develop solutions to these problems and suggest some of the key concepts as relevant to the overall goal of effective automated dynamic testing of software for higher assurance.

We feel that greybox fuzzing provides the most promising approach to Challenge 1 (thorough coverage). Greybox fuzzing uses instrumentation to monitor paths explored by inputs and uses this information to prioritize inputs that uncover new behaviors for further mutation and exploration. To ensure that fuzzing uncovers as many issues as possible, in our approach to the problem, we combine greybox fuzzing with error amplification–that is, transforming the test program in a way that maximizes the chance of raising an alarm in case of undefined or unsafe behaviors. Additionally, to ensure that fuzzing maximally explores the behaviors of the test target, it will become increasingly important to apply grammar-based or structure-aware fuzzing to help focus the evaluation on valid inputs to the test target. (It will be imperative, however, to automate the specification of the grammars/structures as much as possible.)

Binary-only approaches, such as binary editing and software dynamic translation, can address Challenge 2 (lack of source code). To ensure high performance of testing, our approach relies on binary editing.

To address Challenge 3 (running realistic applications), one can create a component that acts as a harness for a wide variety of applications. In our approach, we chose to create a generic harness mechanism that is able to serve data on many different input channels and configure this harness with a manifest. We create such a manifest automatically by observing a few executions of the test target.

Many safety- and security-critical systems are in the form of embedded devices, cyber-physical systems, and other less-studied and less-tested platforms. Generally, automated dynamic testing cannot be executed on such platforms and analysts have relied on complex simulations/emulations to validate system behavior. To address Challenge 4, we are investigating the ability to evaluate components of embedded systems in isolation, for example, by extracting components (such as a web server built into a device) out of an embedded binary and performing extensive fuzzing of the component in an emulation environment. (Note that this requires handling attempted accesses to missing hardware interfaces and system/library calls.)

4 Summary

Higher assurance for the software supply chain requires more automation and more coverage of the possible issues. Automated coverage-driven dynamic testing based on greybox fuzzing and error amplification offers the promise of large scale, easy-to-deploy dynamic testing capable of discovering many subtle flaws. To benefit the common case of software delivery without source code, as well as the maintenance and sustainment of legacy systems, it will be important to apply these techniques to binary executables and libraries. To ease adoption, it will be crucial to automate the process of creating harnesses for complex applications. Finally, it will be vital to extend this capability to many platforms, especially embedded systems, which are controlling an increasing proportion of the world's safety- and security-critical devices.