November 2022

Fuzz-testing Stateful libraries by Shadowing and Comparing

Christopher Jonathan Phoenix

Fuzz-testing Stateful libraries by Shadowing and Comparing

ABSTRACT

Complex software libraries, especially those that store internal states, can be difficult to test. Fuzz testing is an automated testing technique that provides random test inputs to a software module to reveal software defects and vulnerabilities. Traditional fuzz testing doesn't aim to validate the correctness of the output; rather, it attempts to uncover exceptional behavior such as crashes. This disclosure describes techniques to fuzz-test a software library, module, or product in a manner that also tests for correctness of output. A shadow, canonical library is written from the same specification as the production library. The shadow library, simpler and clearer than the production library but having its functionality, provides runtime golden outputs. Identical inputs, possibly auto-generated, are fed to canonical and production libraries and their outputs compared to determine correctness of production code.

KEYWORDS

- Software testing
- Fuzz testing
- Canonical implementation
- Stateful library
- Software library

BACKGROUND

Complex software libraries, especially those that store internal states, can be difficult to test. The behavior of a stateful library can depend on the sequence and details of interactions with it, and it can be difficult to write comprehensive tests that exercise every, or even the most frequently traversed, combination of library state and execution path through the library.

Optimally, tests exercise the library fully, but the number of possibilities inherent in a sequence of multiple inputs combined with internal library states means that it is feasible to cover only a tiny fraction of test space.

Fuzz testing is an automated testing technique that provides random test inputs to a software module to reveal software defects and vulnerabilities. The test inputs can potentially be invalid, malformed, or unexpected, such that the range of possible inputs is deliberately increased. Although fuzz testing increases the range of inputs, it is hard to verify the correctness of the outputs. For example, a fuzz test can discover an input combination that causes a crash, but not one that didn't cause a crash but nevertheless produced incorrect outputs. Traditional fuzz testing doesn't fully validate the correctness of the output; rather, it attempts to uncover exceptional behavior such as crashes.

A standard way to evaluate the functionality of a library is with a set of golden output files, e.g., known, correct outputs for given inputs. By definition, a golden output is tied to a specific test input. Golden outputs are also brittle. Multi-step tests can require many near-identical copies of golden outputs, which can be difficult to maintain. Tests effectively turn into change detectors, and a software developer or tester finds it hard to proofread all the output files to make sure that changes in the output are as intended. Often, rather than taking a broken output as a sign of a new problem, the broken output is rubber-stamped by a human.

DESCRIPTION

In contrast to traditional fuzz testing, which aims only to uncover exceptional behavior, this disclosure describes techniques to fuzz-test a software library, module, or product in a manner that also tests for correctness of output.

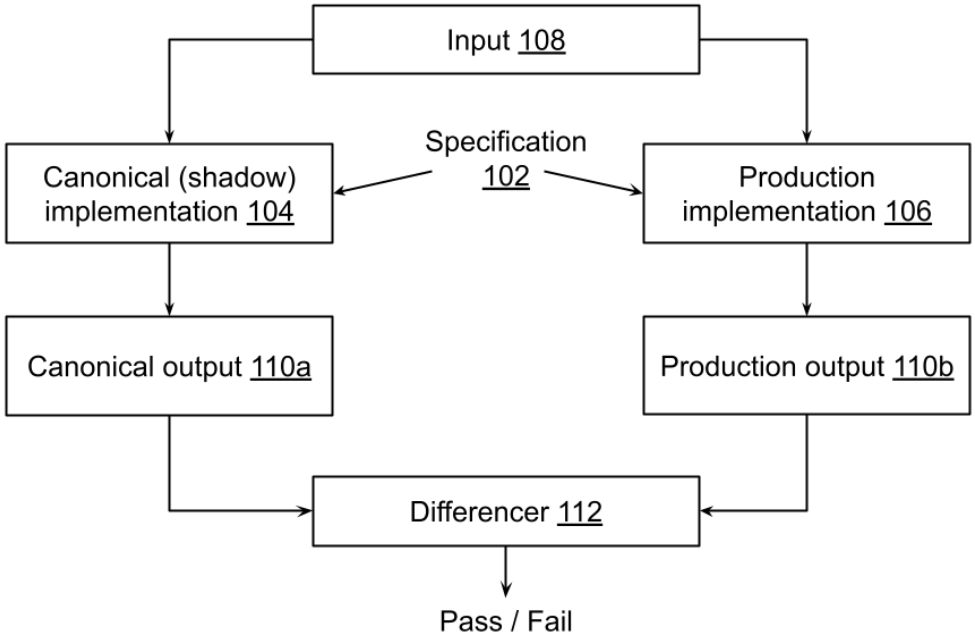

Fig. 1: Fuzz-testing stateful libraries by shadowing and comparing

Fig. 1 illustrates fuzz-testing software, e.g., stateful libraries, by shadowing and comparing to test for correctness of output. Production-quality software libraries aren't merely functional; they are also efficient, they can support multiple languages, they may log diagnostic data, they may include unit tests, etc. Such production requirements often make production-quality software many times bigger than functionality alone would suggest. A production implementation (106) of the library, derived from a specification (102), is the software-under-test. The functionality of the library is implemented in a simple, straightforward way, producing a canonical or shadow implementation (104). The canonical implementation is derived from the same specification that the production software is derived from, e.g., it is implemented from the documentation without referring to production code. The canonical implementation is free of such production requirements as efficiency, multi-language support, etc. It can be straightforward and aim at clarity, e.g., readability. For example, a canonical implementation can use simpler but less efficient algorithms and can be written in a single scripting language. The

canonical implementation can be written by team members different from the team members who wrote the production code.

The canonical and production implementations are supplied with the same test input (108). The output (110a) of the canonical implementation is automatically compared with the output (110b) of the production implementation by a differencer (112). A match between the outputs indicates a passing test, while a mismatch between the outputs indicates a failing test. Effectively, the canonical implementation provides a runtime golden output for a given input sequence, regardless of whether the input is valid or not, or well-formed or not. A large number of sequences of valid and invalid inputs can be generated at runtime by random or combinatorial algorithms, and the output of each sequence can be compared for correctness. Manually created input sequences can also be provided and outputs compared for correctness without needing to generate or maintain golden files.

Comparing the outputs of canonical (shadow) and production implementations, as described herein, can be done at a level of detail decided by the software tester/developer. For example, outputs can be compared at intermediate stages or final stages of execution. Outputs can be compared in depth, e.g., at the level of data-structure elements, or at the status-code or checksum level. Test outputs can be compared during the test as many times as deemed necessary by the tester/developer. Tests can be written in as many steps and operational sequences as deemed necessary by the tester/developer to thoroughly exercise the library. For example, since the behavior of stateful libraries depend on their intermediate internal states, components of the software-under-test can be called in multiple different orders and timings to test across a range of internal states.

Fuzzers can be written that auto-generate inputs. As explained earlier, the canonical library produces golden output at runtime, such that there is no need to maintain static golden files and no need to restrict the input space to pre-written inputs.

Implementing the library in a canonical form from the documentation can point out confusing or ambiguous sections in the documentation. Calls into the library API can be made within the same program, or within stub or puppet programs controlled via inter-process communication (IPC). This is especially relevant when testing libraries that should implement the same functionality in multiple languages.

CONCLUSION

This disclosure describes techniques to fuzz-test a software library, module, or product in a manner that also tests for correctness of output. A shadow, canonical library is written from the same specification as the production library. The shadow library, simpler and clearer than the production library but having its functionality, provides runtime golden outputs. Identical inputs, possibly auto-generated, are fed to canonical and production libraries and their outputs compared to determine correctness of production code.

REFERENCES

[1] “Git repositories on fuchsia,” https://fuchsia.googlesource.com/fuchsia/+/refs/heads/main/src/diagnostics/validator/ accessed Nov. 10, 2022.