Technical Disclosure Commons

Defensive Publications Series

February 13, 2019

Parallelizing tests with pessimistic speculative execution

Tavis Ormandy

Parallelizing tests with pessimistic speculative execution

ABSTRACT

Fuzzy testing or fuzzing is a technique for finding software vulnerabilities. Fuzzing works by feeding quasi-random, auto-generated input sequences to a target program and searching for failures. Fuzzers find inputs that trigger bugs; however, understanding those bugs is easier when extraneous data is removed to the extent possible. Extraneous data is removed by bisection, which is typically a serial procedure, e.g., doesn't advance without knowing the result of the previous step. This makes test case optimization slow, sometimes taking hours to complete while CPU cores sit idle.

This disclosure describes techniques that parallelize the bisection procedure, e.g., by speculatively executing test cases steps ahead of the current position of the bisection procedure. Test results often become known by the time the bisection procedure reaches a certain step, which substantially accelerates testing.

KEYWORDS

- fuzzy testing
- test-case minimization
- delta debugging
- pessimistic execution
- speculative execution
- multi-threaded testing
- distributed testing
- bisection

BACKGROUND

Fuzzy testing (or simply, fuzzing) is a technique for finding software vulnerabilities. Fuzzing works by feeding quasi-random, auto-generated input sequences to a target program and searching for failures. Fuzzers find inputs that trigger bugs; however, understanding those bugs is easier when extraneous data is removed to the extent possible. This removal of extraneous data is called test-case minimization or delta debugging.

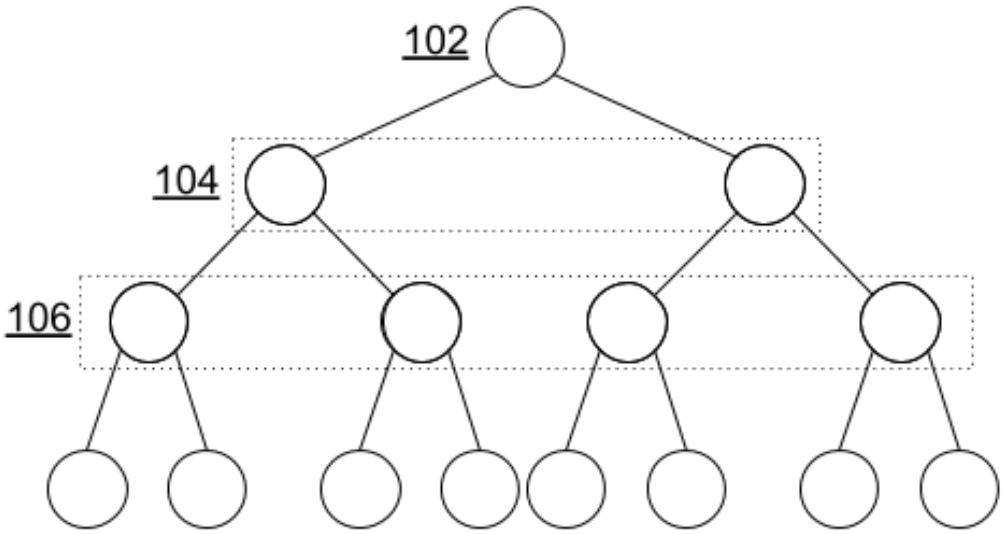

Fig. 1: Bisection to optimize test cases

As shown in Fig. 1, optimization tools typically use a bisection procedure to simplify test cases. For example, if test cases at level 102 pass, then test cases at level 104 are tested. Test cases at level 104 that pass lead to the procedure to move on to testing at level 106 the children of the passing nodes of level 104, and so on. Bisection is an inherently serial process, e.g., it doesn't advance without knowing the result of the previous step. This makes optimization slow, sometimes taking hours to complete while CPU cores sit idle.

DESCRIPTION

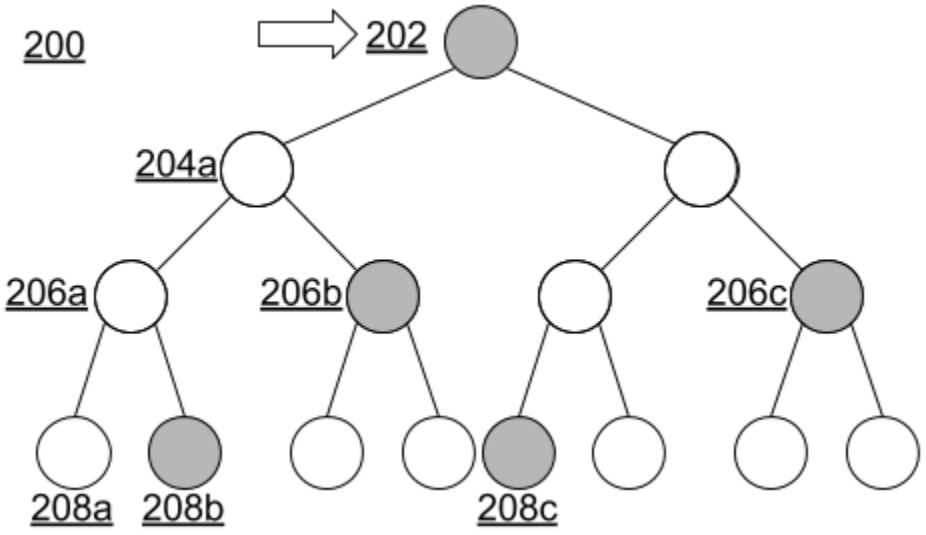

Fig. 2: Parallelizing test-case optimization using speculative execution of future steps

Fig. 2 illustrates accelerating test-case optimization using speculative execution of future steps, per techniques of this disclosure. A binary tree (200) of possible bisection steps is built. In Fig. 2, the bisection procedure is presently at level 202. Even before results of the execution at 202 are known, idle compute cores are used to test steps further down the binary tree e.g., 206b-c, 208b-c, etc. Nodes undergoing such speculative execution are shaded in grey. In many cases, such execution leads to test results of lower-level nodes being already known by the time the bisection procedure reaches them. For example, if the test case 202 passes, and

- if test case 206b fails, then tests of its children are obviated, even before the bisection procedure reaches 206b;
- if test cases 206b and 204a pass, then the bisection procedure can move on to the children of 206b;
- if test cases 204a, 206a, and 208b pass, then the bisection procedure can move on to test case 208a and the children of test case 208b; etc.

The aforementioned procedure is referred to as pessimistic since real workloads are characterized by long series of consecutive failures. The assumption made is that tests are going to fail; the techniques described herein speculatively follow the failure path until proven wrong.

In this manner, the techniques of this disclosure leverage idle compute cores to test software in a distributed, e.g., multi-threaded, manner leading to superior test performance, scalability, and testing efficiency.

CONCLUSION

This disclosure describes techniques that parallelize the test-case optimization procedure, e.g., by speculatively executing test cases steps ahead of the current position of the optimization procedure. The techniques of this disclosure leverage multiple compute cores to test software in a distributed, e.g., multi-threaded, manner leading to superior performance, scalability, and testing efficiency.