Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory,
Stanford University,
Stanford, CA 94305, U.S.A

Abstract. This paper presents a technique that uses code to automatically generate its own test cases at run-time by using a combination of symbolic and concrete (i.e., regular) execution. The input values to a program (or software component) provide the standard interface of any testing framework with the program it is testing, and generating input values that will explore all the “interesting” behavior in the tested program remains an important open problem in software testing research. Our approach works by turning the problem on its head: we lazily generate, from within the program itself, the input values to the program (and values derived from input values) as needed. We applied the technique to real code and found numerous corner-case errors ranging from simple memory overflows and infinite loops to subtle issues in the interpretation of language standards.

1 Introduction

Systems code is difficult to test comprehensively. Externally, systems interfaces tend towards the baroque, with many different possible behaviors based on tricky combinations of inputs. Internally, their implementations tend towards heavily entangling nests of conditionals that are difficult to enumerate, much less exhaust with test cases. Both features conspire to make comprehensive, manual testing an enormous undertaking, so enormous that empirically, many systems code test suites consist only of a handful of simple cases or, perhaps even more commonly, none at all.

Random testing can augment manual testing to some degree. A good example is the fuzz [3,4] tool, which automatically generates random inputs, which is enough to find errors in many applications. Random testing has the charm that it requires no manual work, other than interfacing the generator to the tested code. However, random test generation by itself has several severe drawbacks. First, blind generation of values means that it misses errors triggered by narrow ranges of inputs. A trivial example: if a function only has an error if its 32-bit integer argument is equal to “12345678” then random will most likely have to generate billions of test cases before it hits this specific case. Second, and

* This paper is a shortened version of [1], which was in simultaneous submission with similar but independent work by Patrice Godefroid et al [2]. Our thanks to Patrice for graciously accepting this version as an invited paper.
similarly, random testing has difficulty hitting errors that depend on several different inputs being within specific (even wide) ranges of values. Third, the ability of random testing to effectively generate random noise is also its curse. It is very poor at generating input that has structure, and as a result will miss errors that require some amount of correct structure in input before they can be hit. A clear example would be using random test generation to find bugs in a language parser. It will find cases where the parser cannot handle garbage inputs. However, because of the extreme improbability of random generation constructing inputs that look anything like legal programs it will miss almost all errors cases where the parser mishandles them.

Of course, random can be augmented with some amount of guidance to more intelligently generate inputs, though this comes at the cost of manual intervention. A typical example would be writing a tool to take a manually-written language grammar and use it to randomly generate legal and illegal programs that are fed to the tested program. Another would be having a specification or model of what a function’s external behavior is and generate test cases using this model to try to hit “interesting” combinations. However, all such hybrid approaches require manual labor and, more importantly, a willingness of implementors to provide this labor at all. The reluctance of systems builders to write specifications, grammars, models of what their code does, or even assertions is well known. As a result, very few real systems have used such approaches.

This paper’s first contribution is the observation that code can be used to automatically generate its own potentially highly complex test cases. At a high level, the basic idea is simple. Rather than running the code on manually-constructed concrete input, we instead run it on symbolic input that is initially allowed to be “anything.” As the code observes this input, these observations tell us what legal values (or ranges of values) the input could be. Each time the code makes a decision based on an observation we conceptually fork the execution, adding on one branch the constraint that the input satisfies the observation, and on the other that it does not. We can then generate test cases by solving these constraints for concrete values. We call such tests execution generated testing (EGT).

This process is most easily seen by example. Consider the following contrived routine bad_abs that incorrectly implements absolute value:

```c
0:   int bad_abs(int x) {
1:     if(x < 0)  
2:         return -x;
3:     if(x == 12345678)
4:         return -x;
5:     return x;
6:   }
```

As mentioned before, even such a simple error will probably take billions of random-generated test cases to hit. In contrast, finding it with execution generated testing it is straightforward. Symbolic execution would proceed as follows: