Software Testing, Quality Assurance & Maintenance—Lecture 7

Patrick Lam
University of Waterloo

January 26, 2026

Part I

Fuzzing

Some JavaScript Code

```
function test() {
    var f = function g() {
        if (this != 10) f();
    };
    var a = f();
}
test();
```

Huh?

- this code used to crash WebKit (`https://bugs.webkit.org/show_bug.cgi?id=116853`).
- automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Fuzzing effectively finds software bugs, especially security-based bugs (e.g. insufficient input validation.)

Fuzzing Origin Story

- 1988.
- Prof. Barton Miller was using a modem,
  on a dark and stormy night.
- Line noise caused UNIX utilities to crash!

Fuzzing Origin Story Part 2

- he got grad students in his Advanced Operating Systems class to write a fuzzer
  (generating unstructured ASCII random inputs)
- result: 25%-33% of UNIX utilties crashed on random inputs[1]

[1] http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html

(an earlier use of Fuzzing)

- 1983: Apple's "The Monkey[2]"
- Generated random events for MacPaint, MacWrite.
- Found lots of bugs,
  but eventually the monkey hit the Quit command.

- Solution: "MonkeyLives" system flag, ignore Quit.

[2] http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt

How Fuzzing Works

Two kinds of fuzzing:

- mutation-based: start with existing tests, randomly modify
- generation-based: start with grammar, generate inputs

What you do:

- feed randomly-generated inputs to the program;
- look for crashes or assertion errors;
- or run under a dynamic analysis tool (e.g. Valgrind) and observe runtime errors (implicit oracles).

Level 0 Fuzzing

Generation-based testing for HTML5.

Use the regular expression:

```
.*
```

that is: “any character”, “0 or more times”.

Found a WebKit assertion failure:
`https://bugs.webkit.org/show_bug.cgi?id=132179`.
Process:

- Take the regular expression and generate random strings from it.
- Feed them to the browser and see what happens.
- Find an assertion failure/crash.

Hierarchy of inputs: C

1. sequence of ASCII characters;
2. sequence of words, separators, and white space (gets past the lexer);
3. syntactically correct C program (gets past the parser);
4. type-correct C program (gets past the type checker);
5. statically conforming C program (starts to exercise optimizations);
6. dynamically conforming C program;
7. model conforming C program.

Each level is a subset of previous level, but more likely to find interesting inputs specific to the system.

Operate at all the levels.

Mutation-based Fuzzing

Develop a tool that randomly modifies existing inputs:

- totally randomly, by flipping bytes in the input; or,
- parse the input and then change some of the nonterminals.

If you flip bytes, you also need to update any applicable checksums if you want to see anything interesting (similar to level 3 above).

Quote from Fuzzinator author

More than a year ago, when I started fuzzing, I was mostly focusing on mutation-based fuzzer technologies since they were easy to build and pretty effective. Having a nice error-prone test suite (e.g. LayoutTests) was the warrant for fresh new bugs. At least for a while.

As expected, the test generator based on the knowledge extracted from a finite set of test cases reached the edge of its possibilities after some time and didn't generate essentially new test cases anymore.

At this point, a fuzzer girl can reload her gun with new input test sets and will probably find new bugs. This works a few times but she will soon find herself in a loop testing the common code paths and running into the same bugs again and again.[3]

[3] `http://webkit.sed.hu/blog/20141023/fuzzinator-reloaded`

Fuzzing Summary

Fuzzing finds interesting test cases.

Works best at interfaces between components.

Advantages: it runs automatically and really works.
Disadvantages: without significant work, it won't find sophisticated domain-specific issues.