Automated Test-Case Generation: Fuzzing

Testing, Quality Assurance, and Maintenance
Winter 2020

based on slides by
Jakub Kuderski

UNIVERSITY OF WATERLOO

Automated Test-Case Generation

Manual test case generation can be laborious and difficult:

- Takes human time and effort
- Requires understanding the tested code

Alternative: Automated Test Case Generation -- making computer do the work

Fuzzing

Fuzzing -- set of automated testing techniques that tries to identify abnormal program behaviors by evaluation how the tested program responds to various inputs.

> We didn't call it fuzzing back in the 1950s, but it was our standard practice to **test programs** by **inputting** decks of **punch cards taken from the trash.** (...) our random/trash decks often turned up **undesirable behavior**. Every programmer I knew (and there weren't many of us back then, so I knew a great proportion of them) used the trash-deck technique.
>
> Gerald M. Weinberg

Fuzzing

Fuzzing -- set of automated testing techniques that tries to identify abnormal program behaviors by evaluation how the tested program responds to various inputs.

> We didn't call it fuzzing back in the 1950s, but it was our standard practice to **test programs** by **inputting** decks of **punch cards taken from the trash.** (...) our random/trash decks often turned up **undesirable behavior**. Every programmer I knew (and there weren't many of us back then, so I knew a great proportion of them) used the trash-deck technique.
>
> Gerald M. Weinberg

Challenges:

- Finding interesting inputs
- Exploring whole system, not just individual tools or functions
- Reducing the size of test cases
- Reducing duplication -- test cases may exercise the same parts of codebase

REVIEW ARTICLES

Fuzzing: Hack, Art, and Science

By Patrice Godefroid

Communications of the ACM, February 2020, Vol. 63 No. 2, Pages 70-76
10.1145/3363824

Comments

Credit: Irina Vinnikova

Fuzzing, or fuzz testing, is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or fuzzed, inputs.[35] Since the early 2000s, fuzzing has become a mainstream practice in assessing software security. Thousands of security vulnerabilities have been found while fuzzing all kinds of software applications for processing documents, images, sounds, videos, network packets, Web pages, among others. These applications must deal with untrusted inputs encoded in complex data formats. For example, the Microsoft Windows operating system supports over 360 file formats and includes millions of lines of code just to handle all of these.

Back to Top

https://cacm.acm.org/magazines/2020/2/242350-fuzzing/fulltext

Dumb Fuzzers

Black-box testing technique: does not try to reason about tested programs.

Idea: feed random inputs in and monitor tested programs for abnormal behaviors.

Pros:

- Easy to implement
- Fast

```
cat /dev/urandom | tested_application
```

Issues:

- Relies on the ‘luck’ of random input
- May run the same things over and over again
- ‘Shallow’ program exploration

E.g., zzuf

C & C++ Fuzzers

1. American Fuzzy Lop (AFL):
   - Main development at Google in 2013-2015 (Michal Zalewski et al.)
   - Designed to be practical: collection of effective as possible, as simple as possible
   - Comes with a set of command-line tools for monitoring progress, test case minimization, etc.
   - Additional supported languages: Rust, Python
   - Linux, Mac, Windows (via a fork)

2. libFuzzer
   - Part of LLVM's compiler-rt
   - Main development at Google and Apple in 2015-2016 (Konstantin Serebryany et al.)
   - Designed as a part of the LLVM compiler infrastructure
   - Supports other LLVM-based languages, e.g., Rust, Swift
   - Linux, Mac, Windows

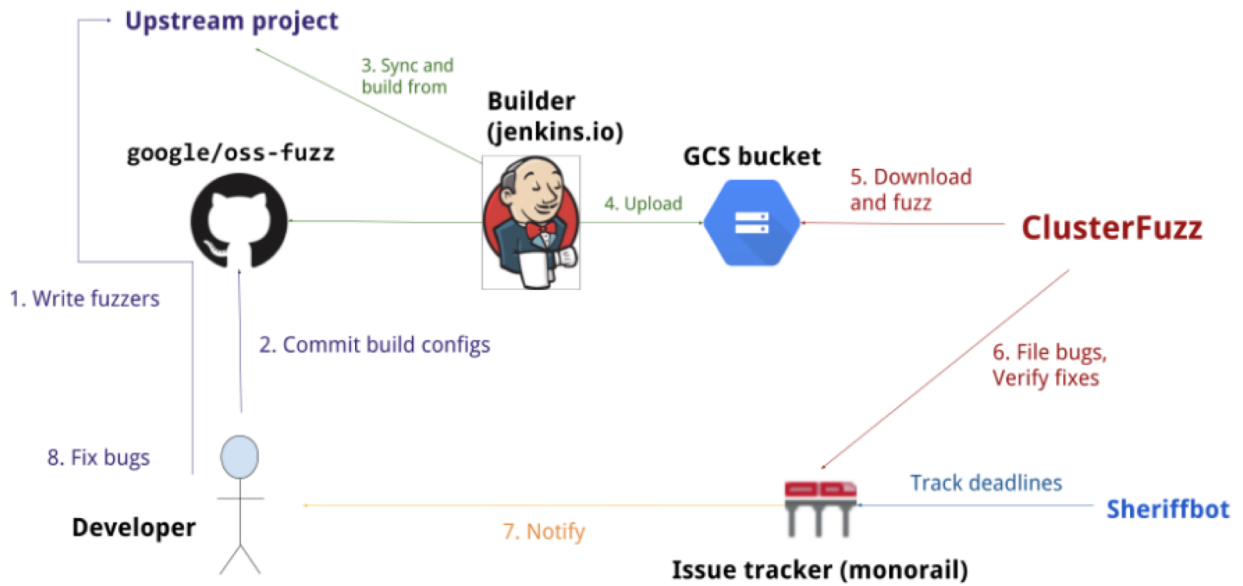
OSS-Fuzz

Project that continuously fuzzes open source project using on a cluster.

We support libFuzzer and AFL as fuzzing engines in combination with Sanitizers. ClusterFuzz provides a distributed fuzzer execution environment and reporting.

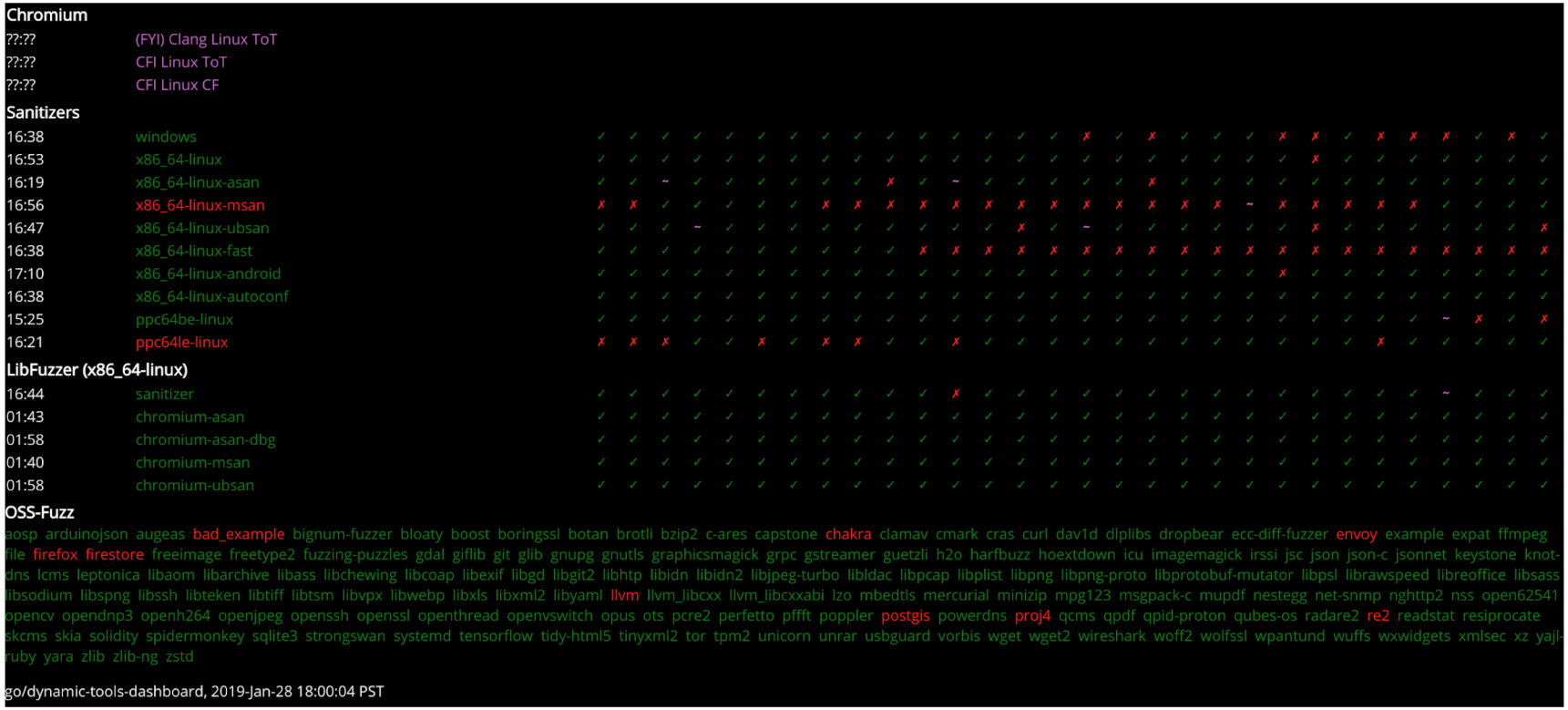
Currently OSS-Fuzz supports C and C++ code (other languages supported by LLVM may work too).

Process Overview



OSS-Fuzz

Project that continuously fuzzes open source project using on a cluster.



http://google.github.io/sanitizers/show_bots.html

Challenge #1: Feeding in inputs

How to take random inputs and make the tested program consume it?

Challenge #1: Feeding in inputs

How to take random inputs and make the tested program consume it?

2 popular open-source fuzzers:

- American Fuzzy Lop (AFL) -- provides wrappers for gcc and clang
  Reads input from files and provides them as STDIN to the tested program.

- LibFizzer -- part of the llvm project, integrated with clang.
  Requires ‘fuzz targets; -- entry points that accept an array of bytes.

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

  The fuzzing engine executes the fuzz target multiple times with different inputs.

Challenge #2: Detecting abnormal behavior

What can ‘abnormal’ mean? We need an oracle.

1. Crashes
2. Triggers a user-provided assertion failure
3. ‘Hangs’ -- execution takes longer than anticipated
4. Allocates too much memory

Challenge #2: Detecting abnormal behavior

What can ‘abnormal’ mean? We need an oracle.

1. Crashes
2. Triggers a user-provided assertion failure
3. ‘Hangs’ -- execution takes longer than anticipated
4. Allocates too much memory

Early crash detection -- use sanitizers:

- Address Sanitizer
- Thread Sanitizer
- Memory Sanitizer
- Undefined Behavior Sanitizer
- Leak Sanitizer

Challenge #3: Ensuring progression

How can we know that fuzzing is exploring more program states over time?

Challenge #3: Ensuring progression

How can we know that fuzzing is exploring more program states over time?
Possible levels of granularity:

1. Instructions (e.g., PC counter position)
2. Lines of source code (using debug information)
3. Statements
4. Control Flow Graph nodes (Basic Blocks)
5. Control Flow Graph edges
6. Control Flow Graph paths
7. Functions

Tracking progression must be very fast

Coverage in AFL: American Fuzzy Lop

Captures branch (edge) coverage by instrumenting compiled programs.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Shared_mem is a 64kB array that represents an approximation of the current program state. Each cell of this array is associated with a counter for multiple basic blocks.

Coverage feedback available in 3 modes:

1. afl-gcc / afl-clang, afl-g++ / afl-clang++ -- wrappers around C/C++ compilers Instrumentation implemented as assembly-level rewriting
2. afl-clang-fast, afl-clang-fast++ -- smarter compiler-level instrumentation Up to around 2x faster.

Coverage in LibFuzzer

Modular: many possible sources of compiler-level coverage instrumentation and mutators:

- Tracing branches, basic blocks, functions
- Optional inline 8-bit counters
- Tracing dataflow: cmp instructions, switch statements, divisions, pointer arithmetic

https://llvm.org/docs/LibFuzzer.html

Fuzzing in nutshell

Core fuzzing algorithm:

```
corpus ← initSeedCorpus()
queue ← ∅
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
    end if
end while
```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

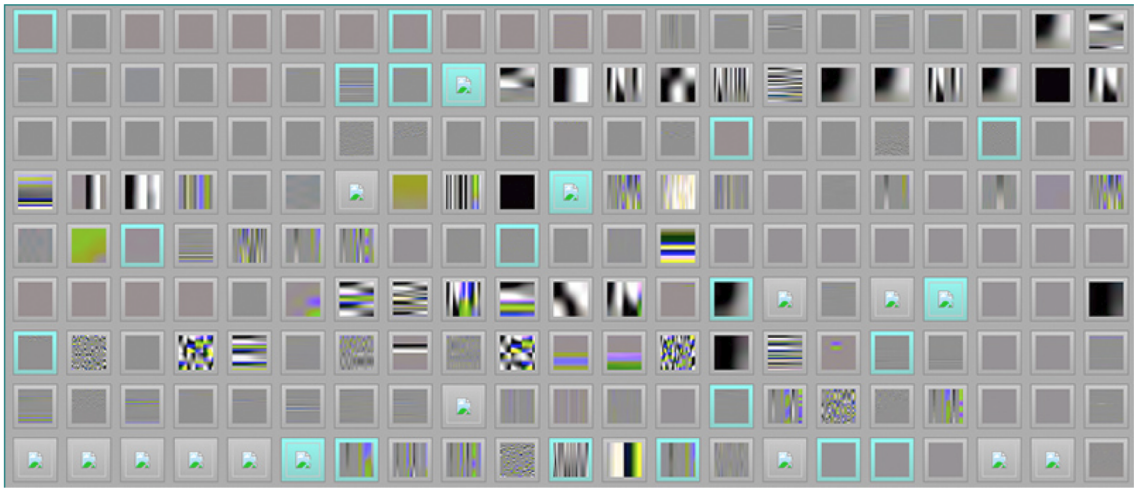
“Pulling JPEGs out of thin air” with AFL

```
$ mkdir in_dir
$ echo 'hello' >in_dir/hello
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg

$ ./djpeg '../out_dir/queue/id:000000,orig:hello'
Not a JPEG file: starts with 0x68 0x65

$ ./djpeg '../out_dir/queue/id:000004,src:000001,op:havoc,rep:16,+cov'
Premature end of JPEG file
JPEG datastream contains no image

$ ./djpeg '../out_dir/queue/id:001282,src:001005+001270,op:splice,rep:2,+cov' > .tmp
$ ls -l .tmp
-rw-r--r-- 1 lcamtuf lcamtuf 7069 Nov  7 09:29 .tmp
```



https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

AFL -- Limitations

1. afl-fuzz is a brute-force tool, and while smart, cannot cover some checks in a large search space (needle-in-a-haystack problems) , e.g.:

```
if (a == “dsaDFDFD”)
    if (b == 143333423442)
        if (c % 234890 == 1999422)
            DoSomethingFunny(...);
```

   Because of that, afl can struggle on formats that contain checksums, e.g., zip, png.

2. afl does not work well on low-entropy inputs, e.g., source code (as very few strings form legal and interesting programs).

Challenge #4: Coming up with interesting inputs

What inputs are likely to trigger a failure?

How to change an existing input to explore more parts of the tested program?

Challenge #4: Coming up with interesting inputs

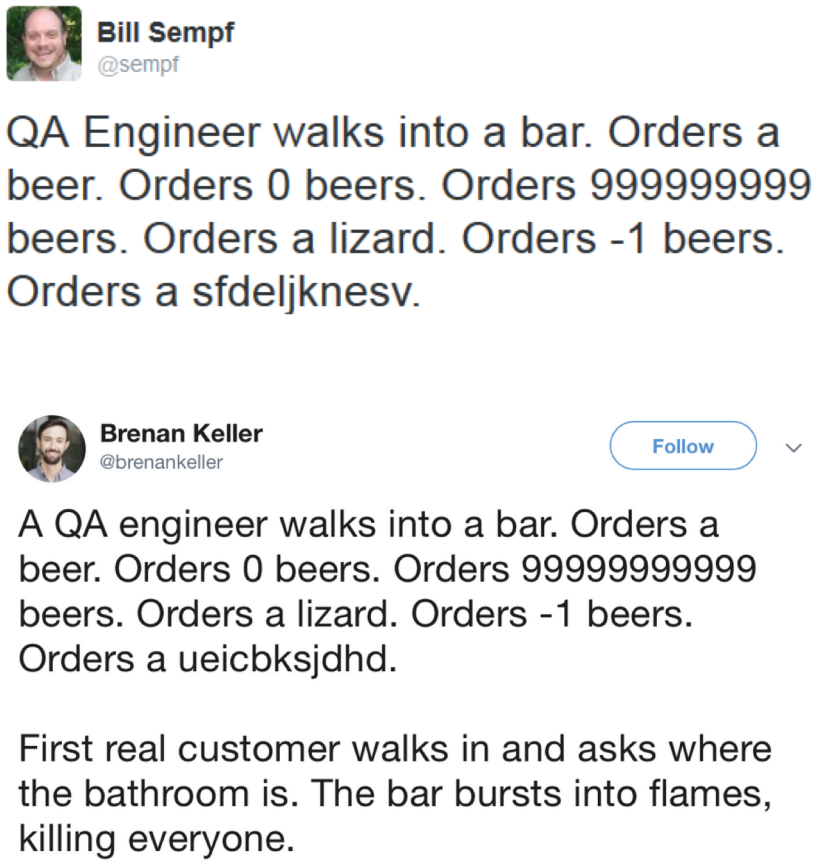
What inputs are likely to trigger a failure?
How to change an existing input to explore more parts of the tested program?

Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

Challenge #4: Coming up with interesting inputs

What inputs are likely to trigger a failure?
How to change an existing input to explore more parts of the tested program?



Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

Brenan Keller
@brenankeller

Follow

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

Challenge #4: Coming up with interesting inputs

Fuzzers start at a provided test-case and keep mutating it.
Examples of mutations:

- Bit flipping: single bit, multiple bits at a time
- Byte flips, byte swaps, byte rotates
- Simple arithmetic: treating groups of bytes as numbers and adding values from predefined ranges: e.g., -128 to +128
- Known interesting integers: e.g., -1, 0, 256, MAX_INT, MAX_INT - 1, MIN_INT
- Combining multiple test-cases together

http://lcamtuf.coredump.cx/afl/technical_details.txt

Challenge #4: Coming up with interesting inputs

Smarter test-case generation: dictionaries.

Some systems expect input in particular formats: XML files, SQL queries, etc.

AFL and LibFuzzer support specifying additional input files: dictionaries with keywords / tokens to use in test-case generation.

How to come up with dictionaries?

Challenge #4: Coming up with interesting inputs

Smarter test-case generation: dictionaries.

Some systems expect input in particular formats: XML files, SQL queries, etc.

AFL and LibFuzzer support specifying additional input files: dictionaries with keywords / tokens to use in test-case generation.

How to come up with dictionaries?

- grep the source code looking for token definitions, files defining grammars
- Provide legal inputs with known parts of the grammar as initial test cases
- Try to make the fuzzer guess the possible tokens

Challenge #5: Speed

What tricks can we use to run a tested program on as many inputs as possible?

Challenge #5: Speed, part 1

What tricks can we use to run a tested program on as many inputs as possible?

1. Avoid paying penalty for start-up time of the tested application: start one copy and clone (fork) when initialization is done
   a. Stop just before main
   b. Stop at a user-specified program point (`__AFL_INIT()` / `LLVMFuzzerInitialize`)

2. Replace costly to acquire resources with cheaper ones, e.g.:
   a. Use a local database instead of connecting to a remote one
   b. Capture inessential network traffic (e.g., using WireShark) and replay it

3. Run many inputs on a single process
   a. Persistent mode in AFL (`__AFL_LOOP(count)`)
   b. Default mode for fuzz targets in LibFuzzer

Challenge #5: Speed, part 2

Minimize the number of test corpuses (test cases) and their size.

- When 2 corpuses result in the same coverage, discard the bigger one
- Take an existing corpus and try to remove parts of it such that the coverage remains unchanged

Further scaling possible by fuzzing in parallel, distributed fuzzing.
E.g., OSS-Fuzz and ClusterFuzz

Fuzzing as an active area of research

1. Automatic fuzz target generation
    a. E.g., with API usage mining

2. Fuzzing for performance
    a. Detecting pathological running time complexity of algorithms

3. Domain-specific fuzzing, e.g.:
    a. Fuzzing compiler optimizations
        i. Satisfiability Modulo Inputs
    b. Checksum-aware fuzzing

4. Improving fuzzing engines with Machine Learning
5. Improving Machine Learning with fuzzing engines

6. Hybrid fuzzing using Symbolic Execution