Fuzzing at Google Today & Tomorrow

Kostya Serebryany <kcc@google.com>

NII Shonan Meeting:
Fuzzing and Symbolic Execution: Reflections, Challenges, and Opportunities
September 2019

Agenda

- Overview of fuzzing at Google
- Fuzz wider
- Fuzz deeper

Scope: Fuzzing C/C++ APIs (source code available)

Not in scope:

- Other languages
  - See cargo-fuzz and go-fuzz
- OS Kernels
  - See syzkaller
- Binaries w/o source code:
  - Useful for offensive security research (hi, Project Zero)
    or for commercial service (hi, MSRD), or for legacy code
  - Counterproductive for 1-st party defensive research and general software quality
- Complete programs with main(), even with source.
  - Counterproductive! Need to Fuzz APIs instead

Sanitizing Google's & everyone's C++ code since 2008

- Testing: ASan, TSan, MSan, UBSan (also: KASAN for kernel)
- **Fuzzing**: **libFuzzer**, Syzkaller, **OSS-Fuzz**
- Hardening in production: LLVM CFI, ShadowCallStack, UBSan
- Testing in production: GWP-ASan
- Hardware-assisted memory safety (Arm MTE)

Testing vs Fuzzing

```
// Test
MyApi(Input1);
MyApi(Input2);
MyApi(Input3);
```

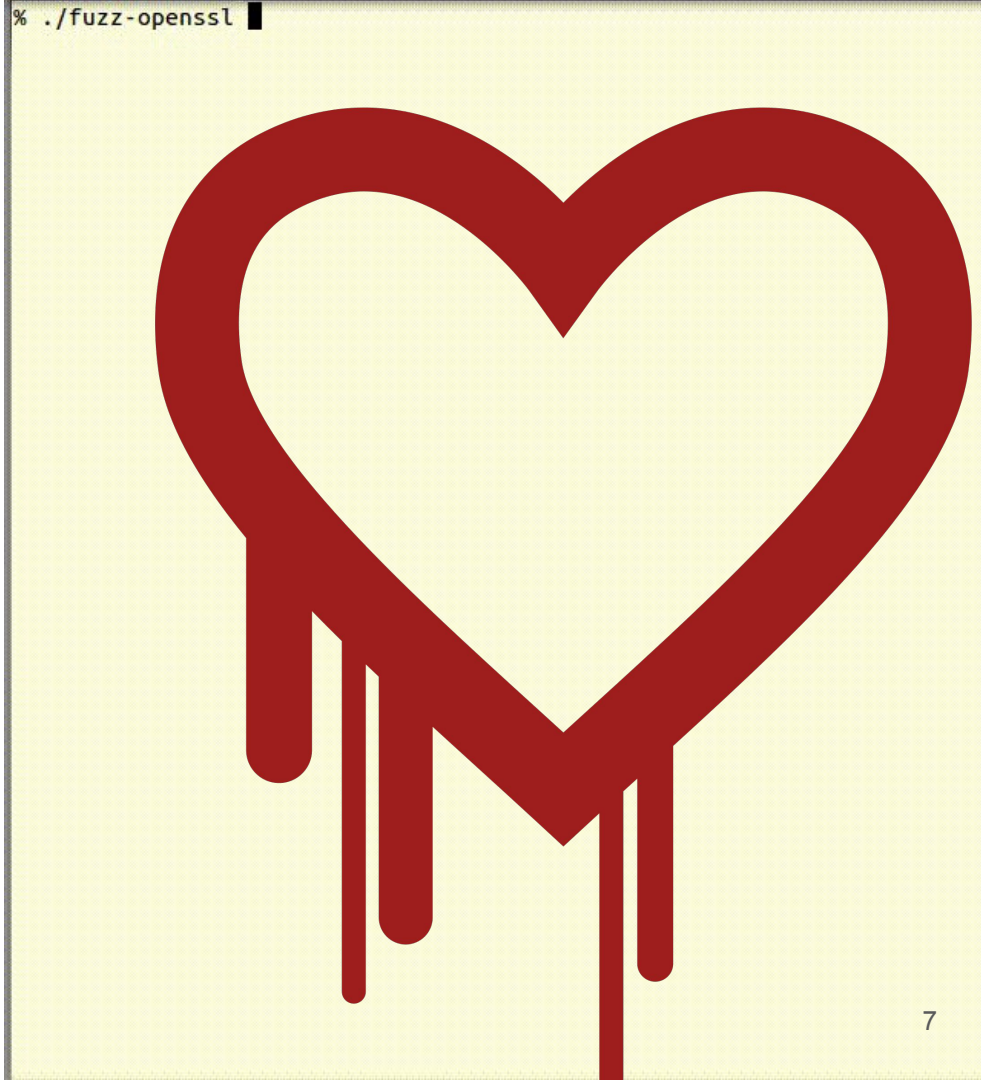
```
// Fuzz
while (true)
   MyApi(
      Fuzzer.GenerateInput());
```

Fuzzing @ Google started gaining traction ~ 2012

- Before ~2012: researchers used their own fuzzers
- First versions of ClusterFuzz (2012-04)
  - Internal, generation-based fuzzers
  - Running generated inputs on the entire Chrome built ASAN
- FFmpeg and thousand fixes (2014-01)
  - First, unguided mutations
  - Later, coverage-guided, ~500 more bugs in FFmpeg
- **Still, only a handful of security researchers**

The Heartbleed

- 2011-12-31: Introduced into OpenSSL
- 😡
- 2014-03: Found independently by
  - Google's Neel Mehta: *code audit*
  - Codenomicon: *specialized fuzzer*
- 2015-04-07 (Hanno Böck):
  - AFL (out-of-process): 6 hours
- 2015-04-09 (Kostya Serebryany):
  - libFuzzer (in-process): 10 seconds



2014-2015: year of realization (for me)

- Fuzzing is not for elites - everyone must fuzz
- Blockers:
  - No easy-to-use-tools
  - Complicated build system integration
  - No continuous fuzzing services
  - No Awareness
  - No motivation
  - ...

libFuzzer

```
bool FuzzMe(const uint8_t *Data, size_t DataSize) {                    // my_api.cc
  return DataSize >= 3 &&
      Data[0] == 'F' &&
      Data[1] == 'U' &&
      Data[2] == 'Z' &&
      Data[3] == 'Z';  // :-<
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) { // fuzz_me.cc
  FuzzMe(Data, Size);
  return 0;
}
% clang -g -fsanitize=address,fuzzer my_api.cc fuzz_me.cc && ./a.out
```

Fuzz Target

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    DoStuffWithYourAPI(Data, Size);
    return 0;
}
```

- Consumes any data: {abort,exit,crash,assert,timeout,OOM} == bug
- Single-process
- Deterministic (need randomness? Use part of the input data as RNG seed)
- Does not modify global state (preferably)
- The narrower the better (fuzz small APIs, not the entire application)

Fuzz Target

- Independent from the fuzzing engine
  - Plug-compatible with most engines today, at least libFuzzer, AFL, honggfuzz
  - Simple standalone main() driver if you need one
- Used for
  - Fuzzing
  - Regression Testing in CI
  - Performance monitoring
  - Any other form of testing (in future)
- Is associated with a corpus (e.g. via BUILD files)
  - Manually selected representative inputs
  - Minimized corpus from fuzzing
  - Inputs for fixed bugs

libFuzzer: coverage-guided fuzzing

- Acquire the initial corpus of inputs for your API
- while (true)
  - Randomly mutate one input
  - Feed the new input to your API
  - **new code coverage** => add the input to the corpus

libFuzzer

- In-process only
- Corpus expansion guided by edge coverage and value profiles
- Mutations (partially) guided by data flow (CMP instrumentation)
- Relies on SanitizerCoverage instrumentation (LLVM-only)
- Supports “custom mutators” (aka mutator plugins)
- Tightly integrated with ASan/LSan/MSan/UBSan
  - E.g. calls LSan after very input where # mallocs != # frees
  - E.g. calls to sanitizer rt to produce better reports
- Corpus merging/minimization, crash minimization/cleansing, parallel fuzzing...
- Part of LLVM distribution

Fuzzing @ Google 2019: finds more vulns than anything else

- Chrome Browser
  - 8 years (In-process API fuzzing: 4 years)
  - 18K bugs (17.5K fixed, 4K: libFuzzer)
  - 4.5K vulnerabilities (99% fixed) (800+: libFuzzer)
- OSS-Fuzz (in-process only)
  - 3 years, 230 projects (and growing), 15K bugs, 13K fixed
  - 3.5K vulnerabilities (over 3K fixed; 2.4K: libFuzzer, 500: AFL)
- Server-side code (libFuzzer, AFL, Honggfuzz, in-process only):
  - 2000+ fuzz targets running 24/7
- ChromeOS, Android, Fucshia, …
- **Self-service**: hundreds (thousands?) of code owners, not security experts
- Fully automated: commit => run => report bugs => track until fixed

Continuous fuzzing infrastructure is critical

- Scaling: compute, storage, fuzz targets, fuzzing engines, sanitizers
- Deduplication
- Bug reporting & tracking
- Fault tolerance
- Testcase minimization
- Regression testing (bisection)
- Statistics for fuzzer performance and crashes
- Friendly GUI
- **Coverage dashboard** ⇐ ask for demo if interested
- Experiment with strategies, new tools, etc

ClusterFuzz: https://github.com/google/clusterfuzz

- The infra behind OSS-Fuzz and Chrome fuzzing
- Runs at 25,000 cores at Google
- Thousands of in-process fuzz targets (libFuzzer, AFL)
- Hundreds of custom generation-based fuzzers
- Linux, Windows, macOS, Android
- Does everything mentioned on the previous slide
  - and even more than that
- Actively maintained and developed, open-source
  - External contributions are welcome :)

Fuzz Wider

Drive further adoption

Google Now:

- We control the build system => made fuzzing builds super easy
- Automated bug finding, reporting, and tracking
- Held FuzzIts, Fuzzathons, and Fuzzing weeks
- Advertized fuzzing in Google toilets worldwide, 4 times
- Fuzzing enforced in some cases by securit reviews
- Fix for a high/critical needs to contain a fuzz target (or explanation why not)
- Patch Reward Program for OSS

You can help:

- Teach fuzzing in college as part of the software development workflow
- Consult OSS and Commercial projects
- Add projects to OSS-Fuzz (and/or MSRD?)
- ???

Fuzz-Driven Development

- Kent Beck @ 2003 (?): Test-Driven Development
  - Great & useful approach (still, not used everywhere)
  - Drastically insufficient for security

- Kostya Serebryany @ 2017: Fuzz-Driven Development:
  - Every API is a Fuzz Target
  - Tests == “Seed” Corpus for fuzzing
  - Continuous Integration (CI) includes Continuous Fuzzing
  - Not specific to C++

Make it simpler

- Easier input split
- Don’t require BUILD file changes
- Automatically suggest targets to fuzz
- ???
- *need your input: for me it’s already simple enough*

Input Split

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
     std::string s1 = <some part of “Data”>;
     std::string s2 = <some other part of “Data”>;
     int i1         = <yet another part of “Data”>
     DoStuffWithYourAPI(s1, s2, i1);
     return 0;
}
```

- Current solutions:
  - Ad hoc: so-so
  - FuzzedDataProvider: ok-ish
  - Protobufs (via LPM): powerful, but too heavy

Proposal: C++ attribute (to avoid build changes)

```
[[fuzz]]

void MyApi(const uint8_t *Data, size_t Size) {…}
```

Proposal: C++ attribute

```
[[fuzz]]
void MyApi(const std::string &s) {…}

[[fuzz]]
void AnotherApi(const std::vector<uint8_t> &v) {…}
```

Proposal: C++ attribute

```
[[fuzz]]

void MyApi(const T &Data) {...}
```

Proposal: C++ attribute

```
[[fuzz]]
void MyApi(const T &Data) {...}
   std::vector<uint8_t> serialize(const T&);
   T deserialize(const std::vector<uint8_t> &);
   T mutate(const T&);  // optional
```

FUDGE: (mostly) automated fuzz target generation

FUDGE: Fuzz Driver Generation at Scale

Domagoj Babić*
Stefan Bucur*
dbabic@google.com
sbucur@google.com
Google
USA

Yaohui Chen*
yaohway@ccs.neu.edu
Northeastern University
USA

Franjo Ivančić*
Tim King*
Markus Kusano*
ivancic@google.com
taking@google.com
kusano@google.com
Google, USA

Caroline Lemieux*
clemieux@cs.berkeley.edu
UC Berkeley
USA

László Szekeres*
Wei Wang*
lszekeres@google.com
wwweiwang@google.com
Google
USA

ABSTRACT

At Google we have found tens of thousands of security and robustness bugs by fuzzing C and C++ libraries. To fuzz a library, a fuzzer requires a *fuzz driver*—which exercises some library code—to which it can pass inputs. Unfortunately, writing fuzz drivers remains a primarily manual exercise, a major hindrance to the widespread adoption of fuzzing. In this paper, we address this major hindrance by introducing the FUDGE system for automated fuzz driver generation. FUDGE automatically generates fuzz driver candidates for libraries based on existing client code. We have used FUDGE to generate thousands of new drivers for a wide variety of libraries. Each generated driver includes a synthesized C/C++ program and a corresponding build script, and is automatically analyzed for qual-

1 INTRODUCTION

Fuzzing has emerged as one of the most effective testing techniques for discovering security vulnerabilities and reliability issues in software. The idea behind fuzzing is simple: the fuzzer executes programs with randomly generated inputs, and monitors their behavior for invalid operations, such as memory corruption issues. Recent advancements in fuzzing technologies, such as coverage-guided fuzzing [22, 32, 42], have enabled fuzzing to reach even deeper program paths and uncover significantly more bugs.

The success of fuzzing has led to significant adoption in the industry, and the emergence of services providing continuous fuzzing for open source and commercial software. For example, Google has developed continuous fuzzing infrastructures to test the security

Fuzzing is not only about finding crashes

- Differential fuzzing
  - Anything that has two implementations and a way to compare outputs: crypto, compression, rendering, …
  - assert(OptimizedFoo(InputData) == ReferenceFoo(InputData))
  - CVE-2017-3732, a carry propagating bug in OpenSSL
- Self-differential fuzzing
  - Single API, but different ways to get from A to B
  - assert(2*X == X + X); // for a bignum class
- Round-trip fuzzing
  - assert(Uncompress(Compress(Input)) == Input)

Fuzz Deeper

We can’t improve what we can’t measure

- Time to retire LAVA & CGC
  - Too small & artificial
  - Benchmarks with main() are counterproductive
- github.com/google/fuzzer-test-suite
  - ~20 fuzz targets based on fixed revisions of real OSS projects
  - Varying difficulty: from heartbleed (<10 seconds) to CVE-2017-3735 (5 CPU years)
  - Main goal: find more coverage
  - Infra to run on GCE and compare results
  - Was used to measure and prove several improvements in libFuzzer
  - But: still too few benchmarks and already requires lots of CPU
- Future: continuous A/B benchmarking as a side effect of continuous fuzzing
  - If/when implemented in ClusterFuzz will benchmark on 25K cores

New signals for corpus expansion

- Boolean edge coverage (everyone)
- Edge counters (like in AFL) - unconfirmed by benchmarking
- Value profiles: for ‘A < B’ use HammingDistance(A, B) as a signal
  - Works like magic on tiny tests
  - Improves results sometimes, but blows up the corpus. Need to tune further.
- Stack depth instrumentation - limited usefulness for stack overflows.
- ???

Guiding the mutations

- Data Flow Substitution
  - During execution: observe CMP(A,B)
  - During mutation: find A in the input and replace with B
- DFT-based fuzzing
  - Data Flow Trace via taint tracking (DFSan)
  - “What bytes of the given input flow into a given CMP”?
  - Automatically detect which function to focus on (e.g. based on coverage and input frequency)
  - During mutation use DFT to choose which bytes to mutate
  - Again: works magically on tiny tests. Unconfirmed on large ones, needs tuning.
- Dictionaries (meh, but still works well)

- ML maybe? Reinforcement learning looks promising

Feedback to and from user

- To user: is my fuzz target already good?
  - Determinism
  - Performance / RAM consumption
  - Shallow bugs? OOMs? Timeouts?
  - Seed corpus coverage, Full corpus coverage
  - Coverage discoverability (from empty corpus)
- From user:
  - More seeds, where to focus, ???
- Annoy the user until ~~morale~~ coverage improves

Structure-aware fuzzing

- Input type is a complicated data structure
  - Compressed? Encrypted? CRCs? TLV?
- Bitflips: valid in ⇒ invalid out (majority of mutations)
- Structure-aware: valid in ⇒ valid out
- For tree-like structures:
  - Mutate the leaf data fields
  - Mutate the tree structure while keeping it valid
- libFuzzer: delegate mutation to the user via “custom mutators”

Fuzzing Protobuf Consumers

- https://github.com/google/libprotobuf-mutator
  - Takes a proto object in memory and applies one local mutation
  - Integration with libFuzzer via a “custom mutator”
  - Mutations of the tree structure (add, remove, shuffle, splice subtrees)
  - Mutations of the data fields via callbacks to libFuzzer
  - Example: [.cc], [.proto]

- Can be used for other input types
  - Create MyType.proto
  - Concert proto to MyType

Example: sqlite3

- Sqlite3: strong focus on quality and security
  - “100% branch test coverage in an as-deployed configuration”
- Fuzzed on oss-fuzz for 2+ years, all 100+ bugs fixed (usually, within one day)
- New structure-aware fuzzer sqlite3_lpm_fuzzer: 30+ new bugs

http://crbug.com/940205

CREATE TEMPORARY TABLE Table0 (Col0 INTEGER, PRIMARY KEY(Col0 COLLATE RTRIM), FOREIGN KEY (Col0) REFERENCES Table0); ALTER TABLE Table0 RENAME Col0 TO Col0;

```
==40263==ERROR: AddressSanitizer: heap-use-after-free on address 0x03b054e0 at pc 0x005c0948 bp 0xbff21e08 sp 0xbff21e04
READ of size 1 at 0x03b054e0 thread T0
  #0 0x5c0947 in renameTokenCheckAll sqlite3.c:102755
  #1 0x530c26 in sqlite3RenameTokenMap sqlite3.c:102779
  #2 0x52df05 in sqlite3ExprListSetName sqlite3.c:98359
  #3 0x54a575 in parserAddExprIdListTerm sqlite3.c:147491
```

Summary

- Good progress with Fuzzing, esp. after ~2015, but not enough
  - All Chrome and OSS-Fuzz data is public, please take a look
- Need to Fuzz wider:
  - Drive further adoption with carrot (mostly) and stick (little bit)
  - Promote as a general testing technique, not just securty (fuzz-driven development)
  - Remove roadblocks, make fuzzing as easy as breathing (e.g. no BUILD file changes)
  - (Semi-)automatically discover fuzz targets
- Need to Fuzz deeper:
  - Large scale non-artificial benchmarking as a byproduct of actual fuzzing
  - Find new signals for corpus expansion and for guiding the mutations
  - Improve the User ⇔ Fuzzer feedback loop
  - More structure-aware fuzzing