Top N challenges of "deep" fuzzing

Kostya Serebryany <kcc@google.com>

FuzzCon Europe, September 2019

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

- Testing: ASan, TSan, MSan, UBSan (KASAN, etc for kernel)
- **Fuzzing**: libFuzzer, Syzkaller, OSS-Fuzz, Libprotobuf-mutator
  - Fuzzing At Google Today And Tomorrow (Shonan 2019-09)
  - Also: building a specialized fuzzer for a proprietary system
- Hardening in production: LLVM CFI, ShadowCallStack, UBSan
- Testing in production: GWP-ASan
- Hardware-assisted memory safety (Arm MTE)

“Deep” fuzzing is not the most important

- How to define “Deep” fuzzing?
  - Find more bugs?
  - Discover more control flow edges? More code paths? More data flows?
  - More “**what else**”?
- More important to Fuzz:
  - Wide: apply fuzzing to more code
  - Often: fuzzing as part of CI, starting with pre-submit
  - Incrementally: focus on the recently changed code
  - Early: design software with fuzzability in mind (fuzz-driven-development)
  - Naturally: design programming languages with fuzzability in mind
  - Young: fuzzing in CS education
- Still, “deep” is interesting
  - Lots of existing fuzz targets. Code owners need to stay ahead of adversaries
  - Fun research

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

Seed corpus

- File formats: crawl the web
- Use corpus from other fuzz targets
  - Crash Windows USB via Fuzzing Linux USB
  - OSS-Fuzz: corpus of one SSL implementation crashes another. Same for font libs.
- Monitor the live system, scavenge “interesting” inputs
  - Choosing what’s “interesting” with a non-instrumented build; or instrumented build in prod
  - Privacy issues, etc
  - How to automate?
  - Better integration of fuzz targets and production code
- Feedback loop from production bugs, e.g. GWP-ASan
  - Some early one-off success cases, but no automation

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

while (true)

- Frequent question: when to stop fuzzing?
- Frequent answer: never
  - More time => more chance to find something new
  - Tools evolve
  - Code evolves
- Diminishing returns after some point
  - *Assuming the code/tools don't change*
  - How do we know when to stop?
  - And when to restart?

- With 350+ projects and growing, OSS-Fuzz starts to cost quite a bit

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

Choosing inputs to mutate / crossover

- k inputs in the corpus
- compute W(i) (i=1..k), the weight of i-th input

- Naive: uniform W(i) = 1 / k
- Naive (libFuzzer): prefer most recent, W(i) ~= 2 * i / (k*k)
- Less naive (Entropic): favor inputs with “infrequent” control flow edges

- Open question: is this important?
  - Entropic vs libFuzzer shows considerable improvement in short runs (hours, days)
  - No sign of improvement in long runs (weeks, months)

- Same problem for choosing pairs/tuples for crossover
  - I’m not aware of research on crossover!

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

Mutation: structure-aware

- Input is not a bag of bytes, but a highly structured input
  - Syntax tree? Graph? Compressed? Encrypted? With checksums?
- Lib**protobuf**-mutator: input is a protobuf (same: thrift, etc)
  - Can describe anything as proto, see e.g. SockPuppet
  - Creating/maintaining protos for non-proto APIs is time consuming
- Syzkaller: input is a sequence of syscalls with constraints
  - Creating syscall descriptions is time consuming
- Open question: how to automate?
  - File format => proto
  - Sequence of API calls => proto

Mutation: guided

- Input and desired new behavior => produce an interesting mutation
- Extreme case: symbolic (concolic) execution
  - scalability challenges

- Limited data flow guidance
  - Capture bytes flowing into conditionals (or memcmp, strcmp, etc)
  - Substitute “left” with “right” in the input
  - libFuzzer, honggfuzz, AFL++, go-fuzz?

- Complete data flow traces
  - Use taint analysis (DFSan) to mark correspondence {input byte} => {conditional statement}
  - Mutate only the bytes that affect the target conditions
  - Early signs that it works great, but far from wide use

Mutation: guided *and* structure-aware?

- Not hard in principle, but not aware of any implementations

Mutation: how to choose a sequence of mutations?

- MOpt: Optimized Mutation Scheduling for Fuzzers
- Is this a taks for ML?

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

Execution: find bugs

- What bugs can we find?
  - Memory safety, assertion failures, resource exhaustion, etc (boring)
  - Logical bugs?
  - ?
- Differential fuzzing: compare implementation A with implementation B
  - Self-differential: assert(2*X == X + X); // for a bignum class
  - Compare two revisions of the same code
- Round-trip: assert(Uncompress(Compress(Input)) == Input)

Execution: collect control flow

- Everyone uses “coverage”, a.k.a. control flow edges
- Compiler options for better signal?
  - Prohibit optimizations that fold control flow or build with -O0?
  - More inlining or less inlining?
  - Function cloning? (Or, context-aware code coverage)
- Someone else’s control flow (differential fuzzing)
  - Nezha fuzzer: multiply my control flow by someone else’s

Execution: data flow

- Turn data flow into more control flow (laf-intel)
  - if (a == b) => if (HighBits(a) == HighBits(b) && LowBits(a) == LowBits(b))

- libFuzzer: value profiles
  - CMP(a, b) => feature[popcnt(a^b)]++;
    - Actively used
  - a[i] => feature[DistanceFromBounds(i)]++;
    - Not enabled, needs more research
- ???

Execution: what else?

- What other signal can we extract from execution?
  - Time / space overhead (PerfFuzz)
  - Stack depth / call depths (libFuzzer)
- Requirements:
  - needs to be convertible into integers
  - needs to be ~ linear (maybe up to quadratic?) by the program size
    - e.g. full execution paths / traces are unlikely to ever work
-

Guided fuzzing

- Acquire the “seed corpus”
- while(true)
  - Choose input(s)
  - Mutate / crossover
  - Execute: find bugs, collect control flow, data flow, *whatever else*
  - Update the corpus: maybe expand, maybe shrink

Every step here needs an improvement!

Corpus expansion

- When to add an input to the corpus, when to evict?
- What is the ideal corpus size?
  - Trade off between preserving information and diluting useful inputs
- Preserve or evict slow / large inputs?

- Ankou Fuzzer: maximize minimal “distance” between corpus elements
  - Distance measured based on control flow
  - Can we add data flow to Ankou?

Misc: fuzzing stateful systems?

- Two typical approaches:
  - Pretend the system is not stateful
  - Reset the state on every input

- Syzkaller is an example (kernels have lots of state), but is highly specialized
  - Can we generalize?

Misc: evaluating fuzzers

- Hard to improve what you can't measure
- FuzzBench: large scale, real-life targets
  - Also: fuzzer-test-suite (deprecated)
  - Meaningful results require lots of CPU
- Evaluate fuzzers while doing useful fuzzing?
  - Moving target, hard to reproduce results
  - Only fuzz targets with saturated corpus
- Evaluate structure aware fuzzers
- Retirement LAVA & CGC is overdue
  - Too small & artificial, benchmarks with main() are counterproductive

Misc: human in the loop

- On a saturated fuzz target, ask the developer to help
  - Visualize the “coverage frontier”, overlay with production coverage
  - Visualize the inputs reaching the frontier, and parts of inputs affecting the frontier conditionals
  - Especially or structure aware fuzzing (e.g. protobufs)
- If there are bugs, or slow / large inputs, help prioritize the fixes
  - Not important in ideal case, where all bugs are fixed. But, …, well, you know

Summary

- Top N challenges of deep fuzzing:
  - Acquiring better seed corpus (e.g. with feedback from production)
  - Guided and structure-aware mutation
  - Smarter corpus expansion
  - Human in the loop

- Lots of interesting research and potential improvements
  - Please help us extend this TODO list :)
    - github.com/google/fuzzing/issues
    - {fuzzing-discuss,libfuzzer@googlegroups.com