Fuzzing Bitcoin Core

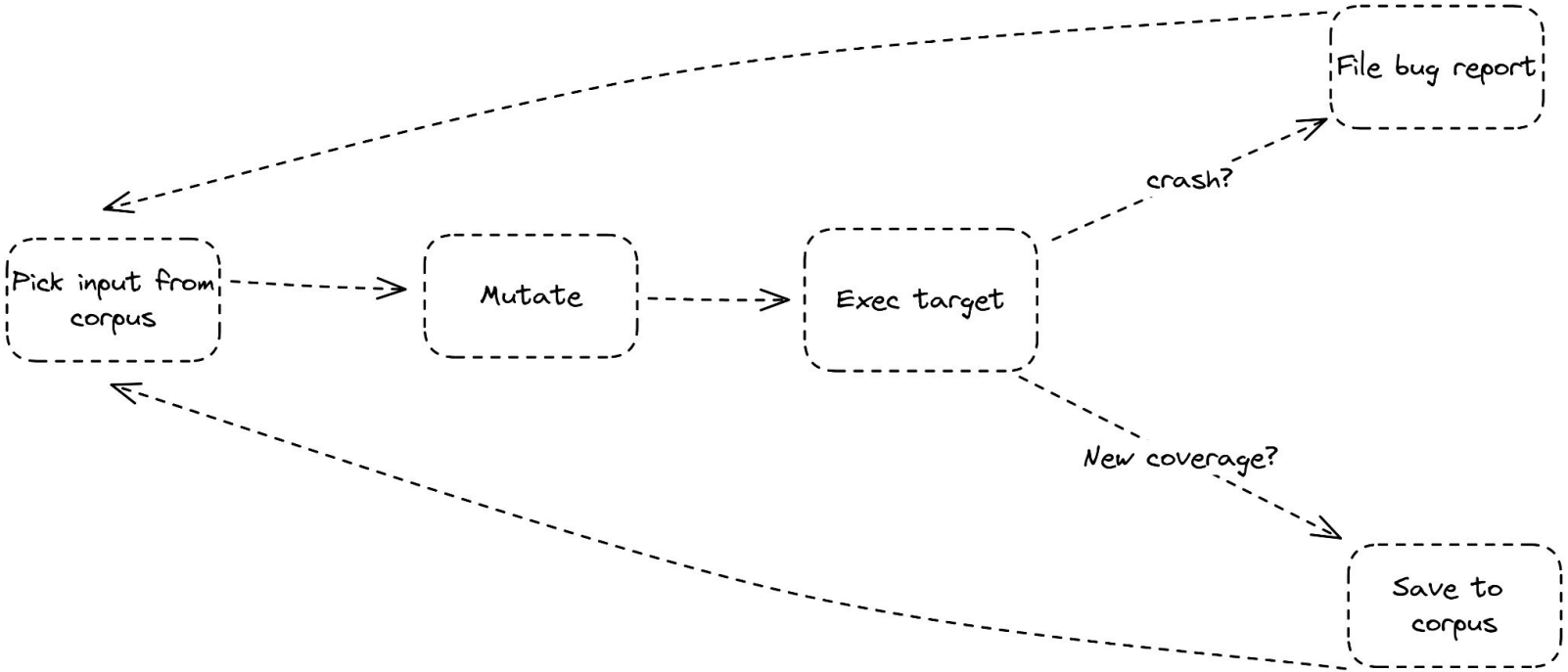
Content

- Fuzzing
  - What is it? Why do it?
  - Coverage guided fuzzers
  - Bug Oracles (Sanitizers, Differential Fuzzing, etc.)
  - Best practices for targets
- Bitcoin Core
  - Fuzzing Infrastructure
  - How/what to contribute

Fuzzing

- Fuzzing: testing code with generated inputs
- Common forms:
  - Mutation based - generate new test cases by mutating existing samples (also known as the corpus)
  - Generation based - generate new test cases based on a model of the input
    - e.g. Fuzzing a C compiler by having inputs generated based on the grammar for C
- Coverage guided fuzzing
  - Extension to mutation based fuzzing
  - Creates feedback loop by extending the corpus with mutated inputs that achieved new coverage
  - Examples: libFuzzer, afl++, centipede, ...

Coverage guided fuzzers



Fuzzing

- Useful when testing …
  - software that takes untrusted inputs (**security**)
  - implementations against each other (**correctness**)
  - high volume APIs (**stability**)
- Not a replacement for regular property based testing (e.g. unit tests)
- Must be done continuously 24/7
  - fuzz targets that are not being executed are not really useful

libFuzzer example

```
struct json_obj* parse_json(const uint8_t *json_str, size_t length) { /** ... */ }

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  parse_json(data, size);
  return 0;
}
```

- `LLVMFuzzerTestOneInput` is the entry to your fuzz target
  - `data` and `size` represent the generated test case as a byte array (passed in by the fuzz engine)
- libFuzzer keeps mutating inputs from the corpus and executing the target until it finds a bug
- “Interesting” inputs (e.g. new coverage) are stored in the corpus →feedback loop
- Fuzz targets are instrumented to help libFuzzer make smarter mutations
  - CMP instruction tracing
  - Shims for byte/string utilities memcmp, strcmp, etc.
  - https://github.com/llvm/llvm-project/tree/main/compiler-rt/lib/fuzzer

How are bugs detected?

- Checking for bugs is harder when inputs aren't fixed
  - You are the oracle when writing unit tests
- Superficial targets might find crashes but they can't find logical bugs
  - e.g. `parse_json` might crash on some weird input but the fuzz target won't report invalid json inputs that pass parsing
- Bug oracles are needed to detect bugs when fuzzing

```
struct json_obj* parse_json(const uint8_t *json_str, size_t length) { /** ... */ }

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  parse_json(data, size);
  return 0;
}
```

Bug Oracles (1/2)

- Assertions
  - Add assertions for assumptions that are internal to your code
- Resource limits
  - e.g. time or memory constraints
- Sanitizers (making C/C++ sane)
  - Undefined-behaviour - detect e.g. integer overflows, out-of-bounds shifts
  - Thread - detect data races
  - Leak - detect memory leaks, malloc/free accounting
  - Memory - detect reads of uninitialised memory
  - Address - detect memory errors, e.g. out-of-bounds heap/stack access, use-after-free

Bug Oracles (2/2)

- Function inverse pairs
  - e.g. encode/decode, encrypt/decrypt
- Differential fuzzing
  - Test 2 implementations of the same thing against each other
  - Pass inputs to both implementations and assert that the outputs are equal
- Null space transformations
  - Only perform mutations that preserve semantics
  - e.g. when testing a compiler, replacing occurrences of x with (x * 1.0) or (x + 1 - 1) should not change the behaviour of the program
- Domain specific checks
  - e.g. A bitcoin block valid under some soft-fork rule should also be valid if the soft-fork is not enforced

Best practices for targets (1/2)

- Avoid non-bug crashes
  - Expect the fuzzer provided inputs to be malformed
  - Write tests for your test utilities
- Verify coverage
  - Make sure your target actually reaches the code under test
  - Work around blockers (checksums, encryption, compression, etc.)
- Determinism
  - Given the same input the target should behave the same
  - Crashes that are not reproducible are annoying
  - Avoid using “actual” randomness (use fixed seeds, mock randomness)
  - Don’t forget to reset global state each iteration (or avoid global state entirely)

Best practices for targets (1/2)

- Performance
  - Fuzzing is a search, the faster the search the better
  - 1000 execs/sec is the benchmark Google recommends
  - Avoid expensive I/O (reading from/writing to disk)
- Keep the scope of targets small
  - Direct the fuzzer to the interesting areas of your code
  - Split into sub-targets for “large” APIs
  - Mock components that are not under test
  - Fuzzing 🤝 Auditing
- https://github.com/google/fuzzing/blob/master/docs/good-fuzz-target.md

Bitcoin Core's Fuzzing Infrastructure

- As of April 2023, we have 195 targets
- Input corpora are maintained at github.com/bitcoin-core/qa-assets
- We are on oss-fuzz
  - ClusterFuzz instance managed by Google to support notable OSS projects
  - "As of February 2023, OSS-Fuzz has helped identify and fix over 8,900 vulnerabilities and 28,000 bugs across 850 projects."
  - 90 day disclosure deadline for bugs (exceptions do apply)
- Contributors run their own infra to generate inputs
  - Hard to quantify how many CPUs are actually running our fuzz targets
  - We don't get a lot of contributions to our corpora😢
  - Not easy to self-host good fuzzing infra

Bitcoin Core’s Fuzzing Infrastructure

- Coverage reports
  - https://marcofalke.github.io/b-c-cov/fuzz.coverage/index.html
  - https://storage.googleapis.com/oss-fuzz-coverage/bitcoin-core/reports/20230213/linux/src/bitcoin-core/report.html
- Fuzz targets are run in CI
  - Uses our input corpora as regression tests
  - Does not generate new inputs
- Fuzzing framework is fuzz engine agnostic
  - Supported: libFuzzer, afl++, hongfuzz, … (basically anything with a byte array interface)
- (Marco)

- On my wishlist: Our own ClusterFuzz instance
  - downside: needs to be maintained
  - up side: can throw money at fuzzing

Fuzzing Bitcoin Core with libFuzzer

```
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin/
$ ./autogen.sh
$ CC=clang CXX=clang++ ./configure --enable-fuzz --with-sanitizers=fuzzer,undefined
$ make
$ FUZZ=process_message src/test/fuzz/fuzz -fork=<number of cores> corpus_dir
```

- Fork mode is great
  - Let's you fuzz on multiple cores
  - Includes a merge step
  - Results in a minimized corpus
- Start with an empty corpus or a seeded one (e.g. from the qa-assets repo)
- Target is specified through the `FUZZ` environment variable
  - `process_message` is a target for fuzzing the processing of a singular p2p message
  - "PRINT_ALL_FUZZ_TARGETS_AND_ABORT=1 ./src/test/fuzz/fuzz"
- Suppressions are required for some sanitizers
  - see "test/sanitizer_supressions/"
- https://github.com/bitcoin/bitcoin/blob/master/doc/fuzzing.md

Merging the corpus into qa-assets

```
$ git clone https://github.com/bitcoin-core/qa-assets
$ FUZZ=process_message ./src/test/fuzz/fuzz -merge=1 qa-assets/fuzz_seed_corpus/process_message corpus_dir
```

- Merging only retains inputs that achieve new coverage
- Open PR to qa-assets with the new inputs
  - New inputs act as regression tests
- Collaboratively growing a corpus accumulates the work that is done

Please contribute

- Run the fuzzers & contribute inputs to our corpora
  - Report sensitive bugs to security@bitcoincore.org (See SECURITY.md)
- Write fuzz targets for uncovered code
  - e.g. The wallet has poor coverage (#27272)
- Improve our bug oracles
- Enforce best practices

Link dump

https://github.com/google/fuzzing/tree/master/docs
https://www.youtube.com/watch?v=UBbQ_s6hNgg
https://www.youtube.com/watch?v=U60hC16HEDY
https://media.ccc.de/v/35c3-9579-attacking_chrome_ipc
https://www.youtube.com/watch?v=NI2w6eT8p-E
https://www.youtube.com/watch?v=S8JvzWDnjc0
https://blog.regehr.org/archives/1687
https://blog.regehr.org/archives/856
https://www.llvm.org/docs/LibFuzzer.html

Input splitting

- Most APIs don’t take a byte array as input
- Common formats (e.g. bolt11 invoices, png images)
  - Desirable if you plan on sharing the corpus between targets or projects
  - Easy to seed
- FuzzedDataProvider
  - C++ helper for dynamically splitting fuzz inputs into various types (provided by llvm)
    - Provides functions to parse fuzz inputs, e.g. ConsumeBool, ConsumeIntegral, ConsumeIntegralInRange
  - Inputs will have a custom serialization format
    - Makes it harder to seed the input corpus
    - Input format can change when the target changes → invalidates the input corpus
- https://github.com/google/fuzzing/blob/master/docs/split-inputs.md