Rich coverage signal and consequences for scaling

Kostya Serebryany
kcc@google.com

July 2023

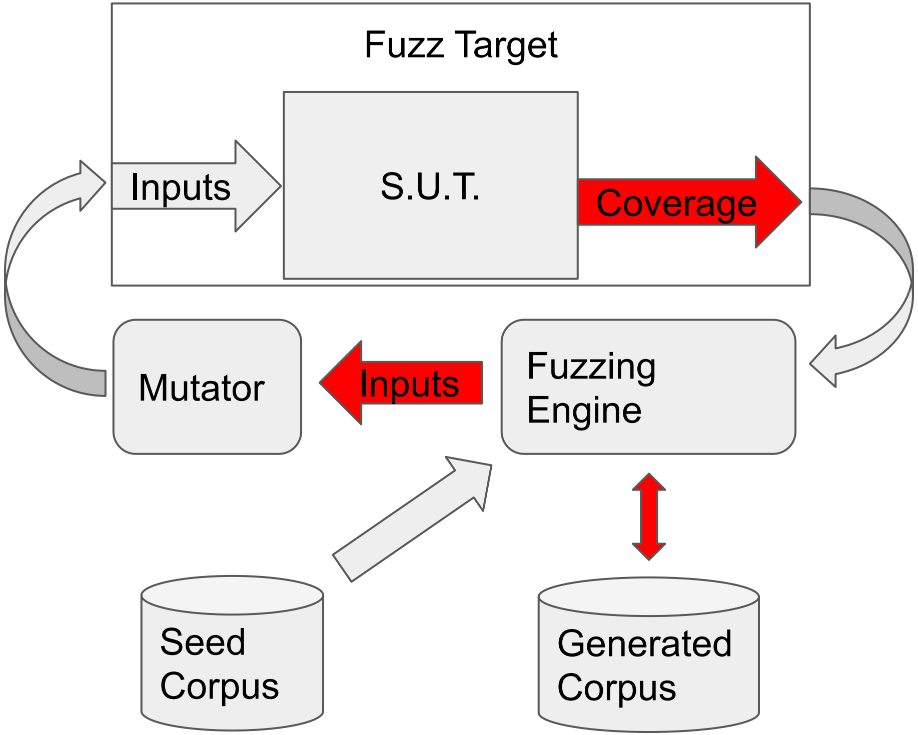
Agenda

- Trivial thoughts on coverage-guided fuzzing
- Case study: SiliFuzz
- Scaling rich coverage signal with the Centipede fuzzing engine

Fuzzing:
generating a maximally diverse
but limited set of test inputs

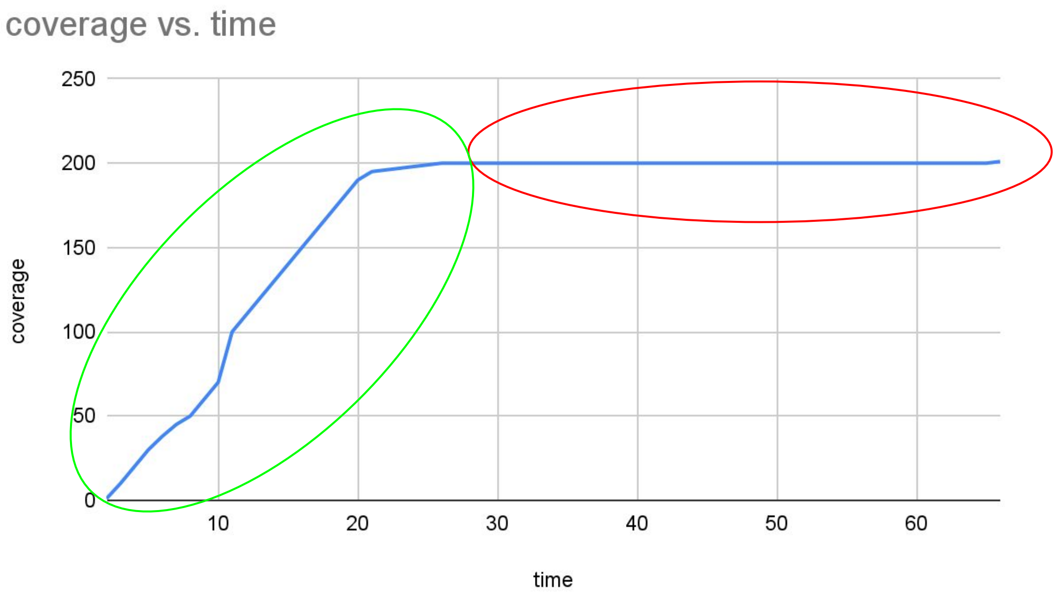
Coverage-guided fuzzing: coverage is the diversity metric

- Inputs with new coverage are added to corpus
- When inputs to mutate are chosen from corpus, coverage is taken into account



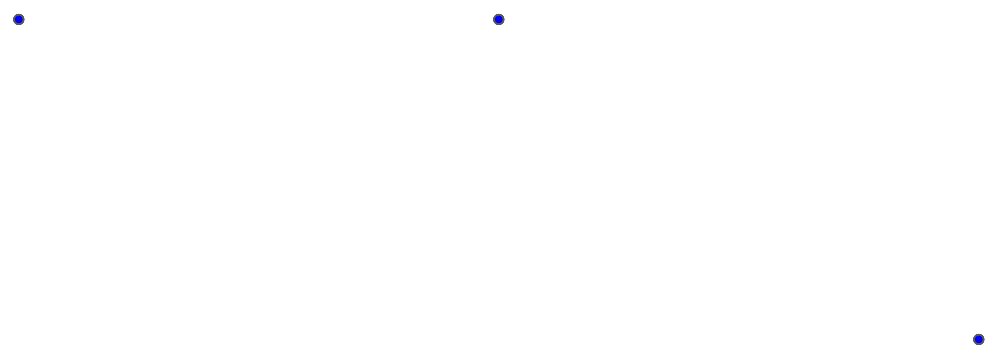
Guided fuzzing is stronger than unguided (*), but…

Guided fuzzing is unguided most of the time



(*) empirically

Initial state: seed corpus



Input in corpus

Early fuzzing: interesting mutants are added to corpus

Input in corpus   Input not in corpus   Input newly added to corpus

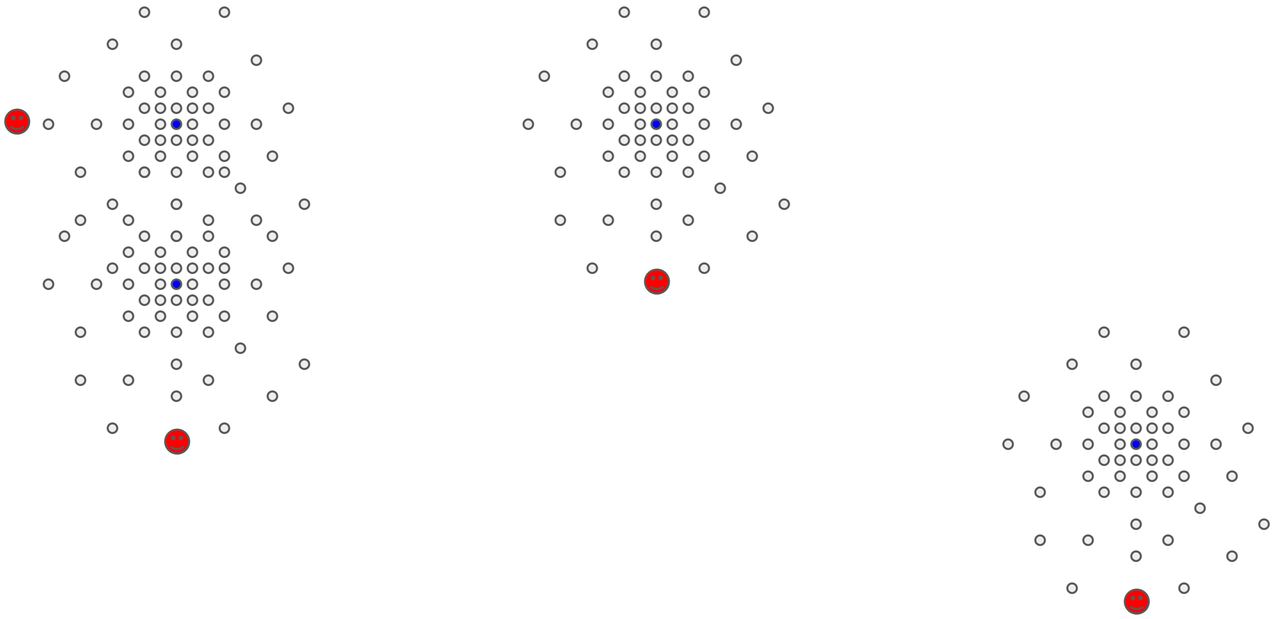
Corpus grows

Input in corpus

Input not in corpus

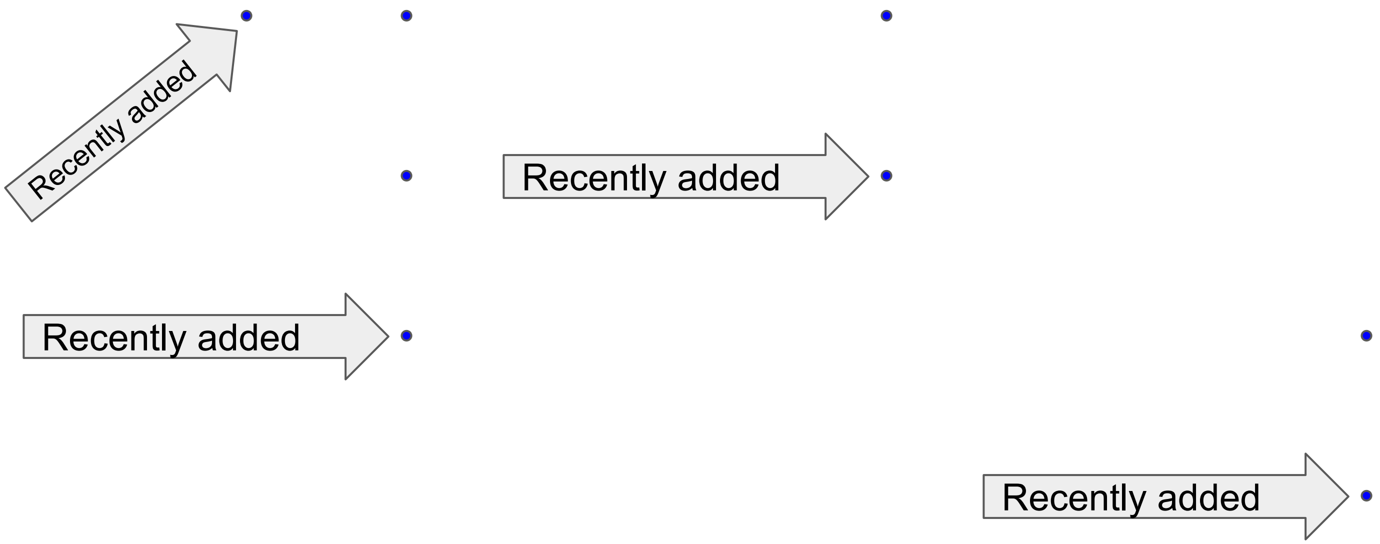
Input newly added to corpus

More fuzzing



Input in corpus   Input not in corpus   Input newly added to corpus

Corpus grows

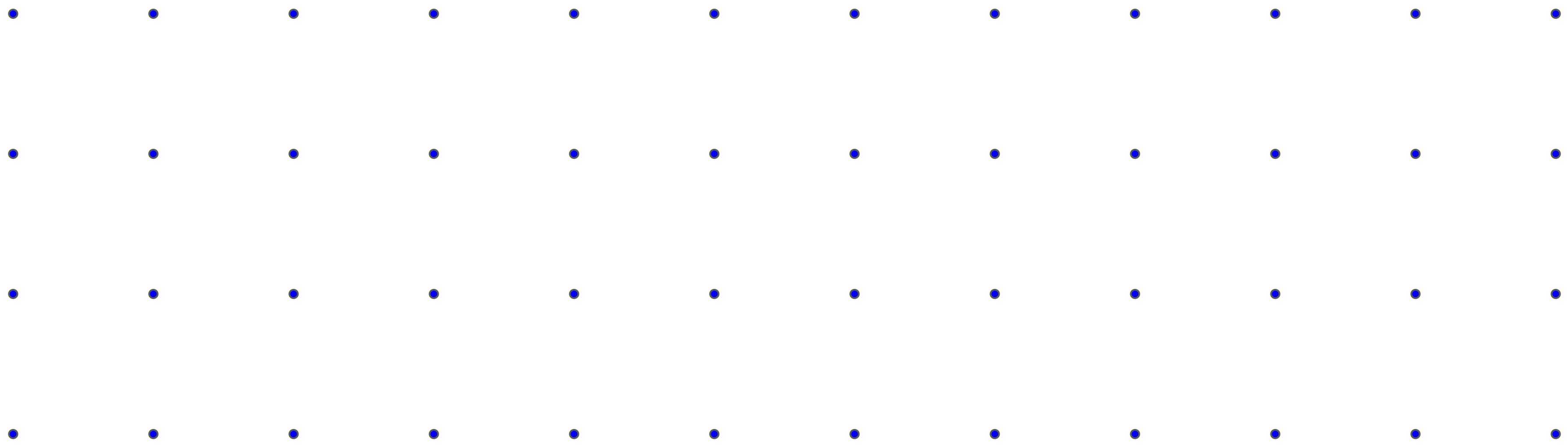


Input in corpus

Input not in corpus

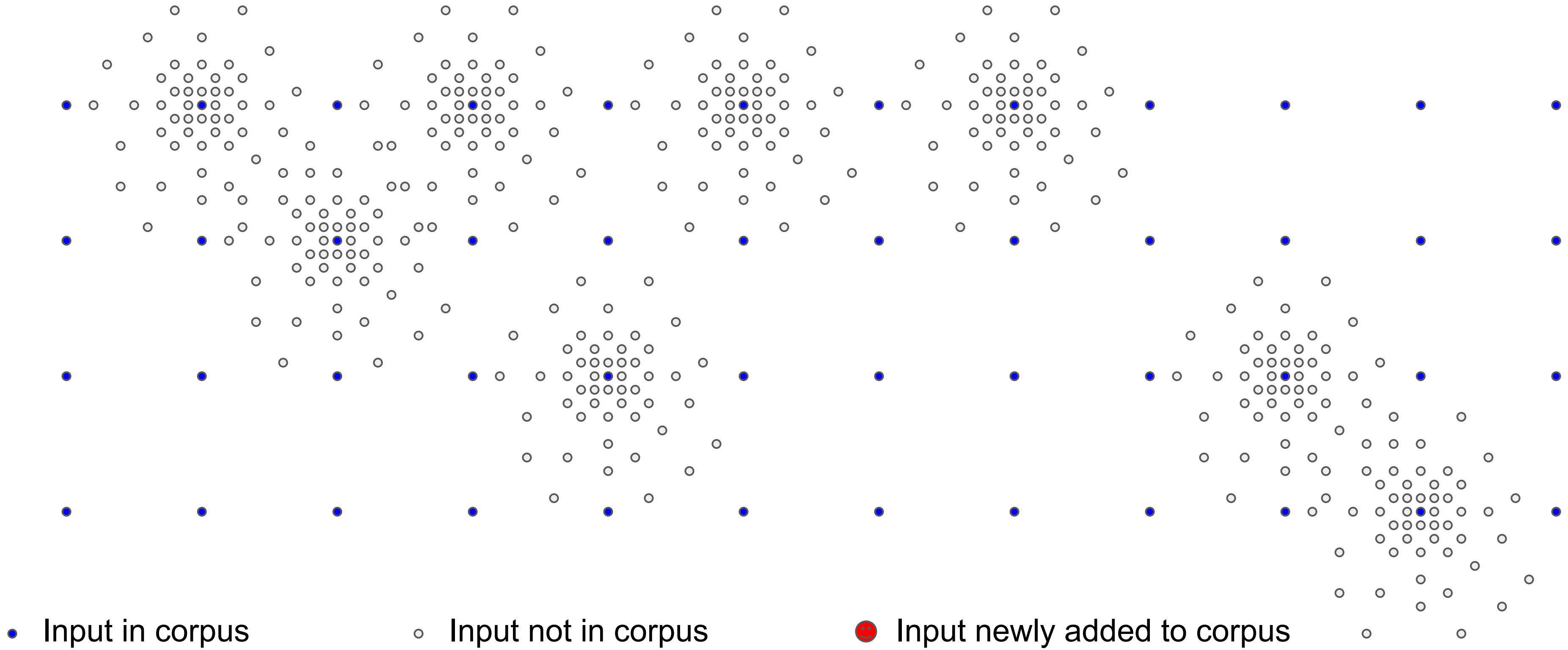
Input newly added to corpus

And grows … until it doesn't



Input in corpus Input not in corpus Input newly added to corpus

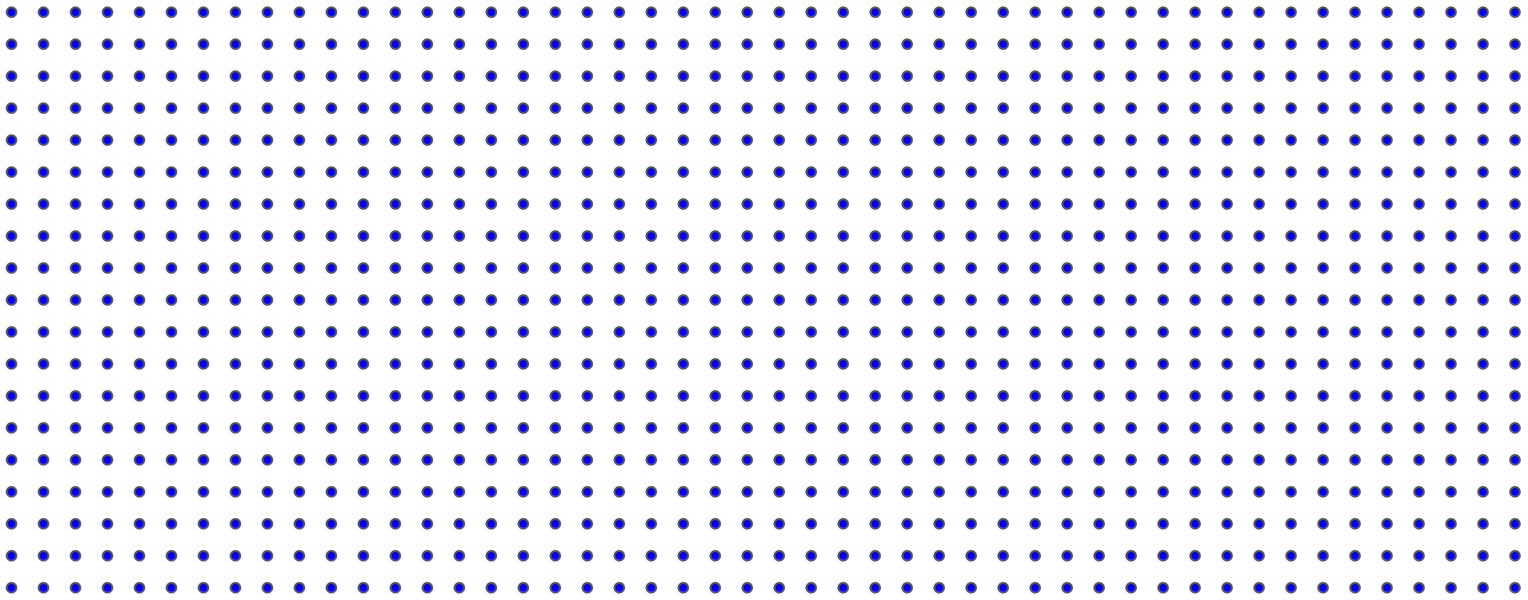
And grows … until it doesn't



Problem: traditional coverage signals are sparse

- Most popular coverage signal: control flow edge
  - O(program size) number of different coverage points
  - O(program size) total corpus elements with different edges
- Other popular signals
  - Counters or Value Profiles - still O(program size)
  - More numerous than edges and already cause scalability problems

Desired: dense coverage signal, but fuzzing still scales



• Input in corpus

Rich (dense) coverage =>
slower runs, larger corpus, more runs =>
more CPU, RAM, and Disk

Case Study: SiliFuzz

SiliFuzz: detects CPU *bugs* and *defects*

- CPU bug: a family of CPUs is affected
  - CVE-2021-26339: A bug in AMD CPU’s core logic may allow for an attacker, using specific code from an unprivileged VM, to trigger a CPU core hang …
- CPU defect: a single physical CPU is affected, often just one core
  - **`imul mem16, %si`** leaves **`%si`** unchanged. Only on one core of one machine.

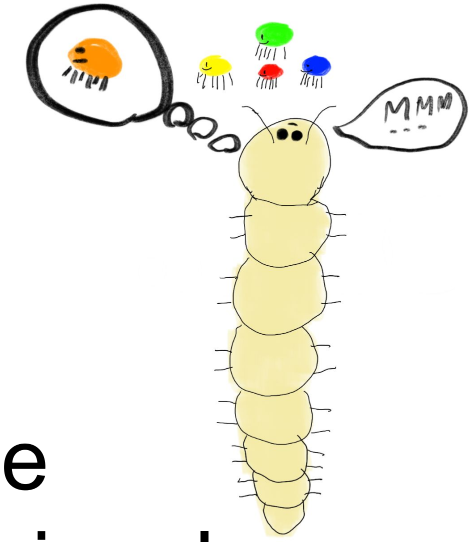
[0] SiliFuzz: Fuzzing CPUs by proxy. arxiv.org/abs/2110.11519
[1] Cores that don't count. dl.acm.org/doi/10.1145/3458336.3465297 (Google 2021)
[2] Silent Data Corruptions at Scale. arxiv.org/abs/2102.11245 (Facebook 2021)
[3] Detecting silent data corruptions in the wild (<link>, Meta 2022)

SiliFuzz: fuzzing by proxy

- Fuzz something that behaves like a CPU (simulator, RTL design, etc)
  - Generates a diverse set of instruction sequences
  - Slow, requires lots of resources
- Run on one machine per family to detect bugs
  - Separate step (but can be combined with fuzzing too)
  - Relatively fast
- Run on all machines in the fleet to detect defects
  - Separate step
  - Huge scale, very expensive and slow

Anecdotal evidence in support for rich coverage signal

- We fuzzed a CPU proxy for ~ 1 year, found a few bugs & defects, saturated
- Added two “rich signals” and found new bugs within a week
- Added another “rich signal” and found yet another bug
- Rich signals increased corpus sizes by 1000x and more, and caused all sorts of scalability challenges



Centipede:
a distributed fuzzing engine
with support for rich coverage signal

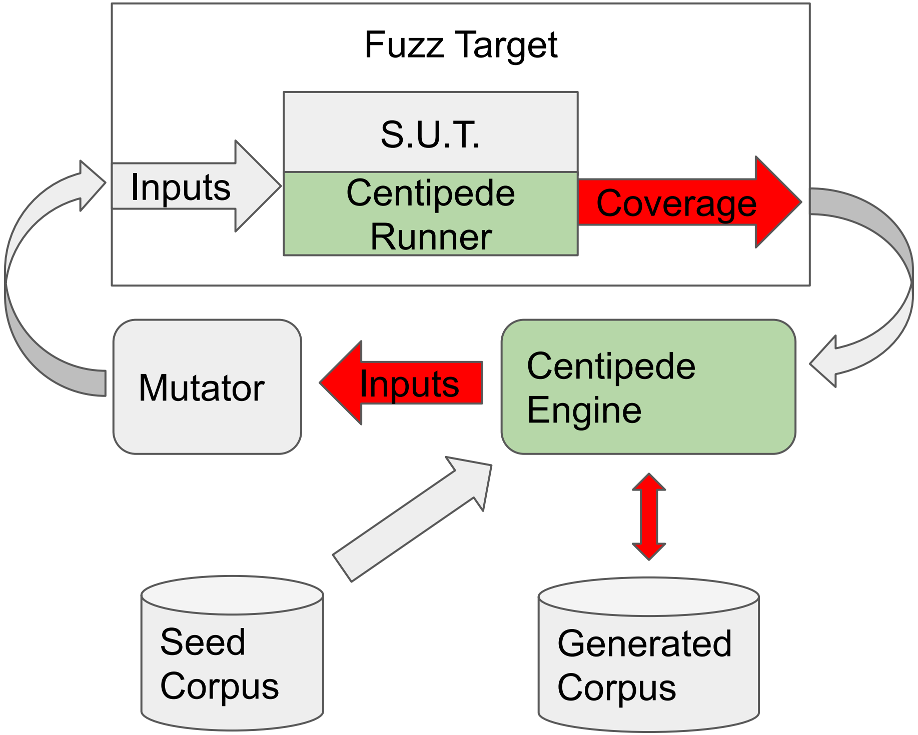
https://github.com/google/fuzztest/tree/main/centipede

Centipede’s goals

- Efficient fuzzing for any kind and size of target, any language
- libFuzzer-compatible
- In- and out-of-process
- Handle rich ‘coverage signal’: 1B+ distinct “features”
- Handle corpus of 1B+ inputs
- Massively distributed
- Stateful: restart != recompute
- User can redefine
    - how to execute the target
    - how to mutate inputs

Centipede: Engine vs Runner

- Runner is linked to S.U.T.
  - Consumes a batch of inputs
  - Reports coverage
  - Can be substituted
- Engine is a separate binary
  - Orchestrates mutation & execution
  - Maintains corpus & coverage



Features and Feature Domains

- *Feature* is a 64-bit integer that uniquely represents some program behavior
  - E.g. “this edge has been executed” or “this constant variable has been accessed at this PC”
- A *feature domain* groups all features of the same type
  - E.g. “control flow edges” or “data flow edges”
  - Currently, domain is a fixed integer range of 2**27
- For every input, the runner reports a set of features to the engine

Feature domains supported currently

- PCs (control flow edges)
- Call stacks: hash of the top N call stack frames
- Paths: hash of the recent N control flow edges
- PC pairs: a pair of control flow edges
- CMP features: up to 64 different values for every CMP
  - Uses call stacks or paths as context
- Edge counters: 8 buckets, similar to AFL/libFuzzer
- Limited data flows: {global memory location} X {where it is read}
- 16 user-defined feature domains: anything you want
  - Example from SiliFuzz: {instruction opcode} X {index of a bit changed in the register state}
- More will be added: limited only by our imagination
  - Full data flows

Shards, state, distributed execution

- File format: one file contains many data blobs, appendable, remote
- Corpus is represented on disk as N shards
  - file per shard with inputs
  - file per shard with {hash(input), features}
- N workers running concurrently (threads, or different machines)
  - Worker appends only to its own shard, peeks into other shards
- On worker restart, reads inputs and features
  - Recomputes coverage only when unavailable

- Each shard represents only a subset of the corpus

Guided Mutation and Execution Metadata

<not covered here>

Corpus management

- Runner: input => array of features
  - O(num_inputs * num_features) bytes, too much
- Engine preserves only some of this data
  - Saturated frequencies for all observed features; a feature observed N+ times is “frequent”
    - O(num_features) bytes
  - Inputs and their “infrequent” features.
    - O(num_inputs * N) bytes
- Chooses what inputs to mutate or evict by assigning weights to inputs
  - less frequent feature => better
  - less frequent domain => better
- Future work: apply ML :)

Corpus distillation (minimization)

- Takes N shards (inputs with their features) and produces distilled corpus
  - Same features, minimized number of inputs
- IO-only task, does not involve executing the target

Centipede vs {libFuzzer, AFL, … }

- With basic coverage: not as tuned, some missing functionality
  - E.g. memcmp / strncmp interceptors
  - But we’ll get there
- With rich coverage: will be seen as weaker on short runs (1-2 days)
  - More breadth than depth, requires more iterations to get to the same edge coverage
  - But saturates later
- Works well on huge targets
  - A CPU model: 75M edges (equivalent), 10K+ concurrent shards

Q&A