Improving Grey-Box Fuzzing by Modeling Program Control Flow

Siddharth Karamcheti, Gideon Mann, David Rosenberg
Bloomberg - CTO Data Science
- Use Machine Learning to Intelligently Hunt Bugs in Programs

- Learn Models of Program Behavior & Exploit Them
Fuzzing - A Background

Fuzzing is a technique for automated software testing.

- Core Idea: Provide programs with unexpected inputs, with goal of finding bugs, maximizing coverage, etc.

Different types of fuzzers and testing tools:

- **Black-Box**: Assume no transparency into program.
  - Random generators (good for testing parsers)
- **White-Box**: Assume lots of transparency (e.g. KLEE)
  - Suffer from path explosion as programs get big
Grey-Box Mutational Fuzzers & AFL

→ Grey-Box Mutational Fuzzers
◆ Mutate existing seed(s) to generate new test inputs
◆ Light instrumentation, to check code paths
◆ Execute test input, add to set of seeds if new behavior

→ AFL is best of the bunch!
◆ Uses heuristics to pick inputs
◆ Randomness for mutations

```plaintext
1: // Core Algorithm for American Fuzzy Lop (AFL)
2: // time: Fixed time window to fuzz (e.g. 24 hours)
3: // queue: Queue of inputs that exercise new code paths.
4: while time has not elapsed do
5:     parent, energy ← pick_input(queue)
6:     for i ∈ range(energy) do
7:         child ← parent
8:         for j ∈ 1 to sample_num_mutations() do
9:             mutation ← sample_mutation()
10:            site ← sample_mutation_site()
11:            child ← apply_mutation(mutation, child, site)
12:         end for
13:     path ← execute_path(child, code)
14:     if (path is new) then queue ← child
15: end for
16: end while
```
Related Work – Improving Grey-Box Fuzzing

AFL relies on **heuristics**, randomness, and scale to find bugs.

Q: Can we be more efficient?

A: Yes, with **data-driven control**!

---

```
1: // Core Algorithm for American Fuzzy Lop (AFL)
2: // time: Fixed time window to fuzz (e.g. 24 hours)
3: // queue: Queue of inputs that exercise new code paths.
4: while time has not elapsed do
5:     parent, energy $\leftarrow$ pick_input(queue)
6:     for $i \in \text{range}(energy)$ do
7:         child $\leftarrow$ parent
8:         for $j \in 1$ to sample_num_mutations() do
9:             mutation $\leftarrow$ sample_mutation()
10:            site $\leftarrow$ sample_mutation_site()
11:            child $\leftarrow$ apply_mutation(mutation, child, site)
12:        end for
13:     path $\leftarrow$ execute_path(child, code)
14:     if (path is new) then queue $\leftarrow$ child
15: end for
16: end while
```

- **AFLFast**
  Böhme et al, 2016

- **Fuzzing by Thompson Sampling**
  Karamcheti et al, 2018

- **FairFuzz**
  Lemieux and Sen, 2017

... and many many more!
Problem: Wasteful Executions

➔ Generating millions -> billions of inputs and executing all of them
  ◆ Fine when program execution is cheap/fast
  ◆ Of billions of inputs generated, maybe only a couple hundred are useful!

➔ Let’s reframe AFL...
  ◆ Break into two phases
    i. Generate: Generate a large number of inputs without execution
       ○ Lightning Fast - just string manipulation
    ii. Execute: Pick some subset of these inputs to execute
       ○ Expensive - depends on the program under test
Q: How do we learn which inputs to execute?

A: Focus on inputs we can't characterize or understand!
**Fuzzing by Modeling Program Behavior**

Given input, predict distribution over program paths, then compute score (entropy of distribution).

Program *Monitored by AFL*

Inputs added to queue if they result in increased code coverage.

Discard

Execute
Learning Prediction Models

→ Learn mappings of inputs (strings) to program paths!
  ◆ For every input AFL executes, it stores the set of control flow graph edges traversed.
  ◆ Treat string input as $x$, set of edges as $y$
  ◆ Fit a model using a classifier of your choice —> just needs to output probabilities!
  ◆ Can update model online! After executing new inputs, train on resulting data!

→ In our work, we keep it simple
  ◆ Featurizer: Bag of Bytes (0 - 255) for encoding input strings
  ◆ Classifier: Logistic Regression
Critical Point: Ranking Inputs by Entropy

➔ We have a model that maps inputs to a distribution over control flow paths
  ◆ What now?

➔ Run generated inputs through our model!
  ◆ Look at the probability distribution over possible code paths
    ● If high-entropy - *crucial input* - means our model isn’t sure
      ○ *Two Benefits to Execution*
        ◆ Could traverse new code path (or hit a bug!)
        ◆ Improve our model by re-training
    ● If low-entropy - *discard* - we know it is redundant
Framework Pseudocode:

1: // Algorithm for AFL + Program Modeling
2: // afl: Instance of AFL for generating/executing inputs
3: // iterations: Fixed number of generation iterations
4: // num_generate: New inputs to generate each iteration
5: // α: Fraction of generated inputs to execute each iteration
6: // queue: Queue of inputs that exercise new code paths
7: // model: Predicts distribution over execution paths for an input
8: // ranker: Given predictions, ranks by entropy values (high - low)
9: for i ∈ range(iterations) do
10:   generated ← afl.generate(queue, num_generate)
11:   execute := []
12:   for g ∈ generated do
13:     execute ← model.predict(g)
14:   end for
15:   execute ← ranker.rank(execute)
16:   for j ∈ range(α· num_generate) do
17:     queue, path ← afl.execute(queue, execute[j])
18:     model ← model.retrain(execute[j], path)
19:   end for
20: end for
We use the DARPA Cyber Grand Challenge Binaries

- Set of 200 binaries released for testing bug discovery + patching
- Each binary has a bug added by a human user (meant to be somewhat “realistic” as opposed to synthetically injected bugs)
- Due to time constraints, we evaluate on a subset of 24 of these binaries
Preliminary Experiments – Baselines

We use the following procedure across all experiments:

- Run AFL for 3 minutes to warm-start/populate queue.
- Start the following 4 Strategies using the resulting queue:
  1. **AFL**: Generates an input then immediately executes
  2. **Batch-AFL**: Meant to mimic our program modeling procedure.
     - Generate 50,000 inputs, execute the first 5000 (slightly different than standard AFL due to the heuristic sampling seed inputs)
  3. **Random Batch-AFL**: Generate 50,000 inputs, select random 5000
  4. **ML/Logistic Regression**: Generate 50,000, use model to pick 5000
Results on 24 CGC Binaries
True Wisdom is Knowing What You Don’t Know

–Confucius
Summary

Questions: Email me @ sidd.karamcheti@gmail.com

Thanks for listening!