ADAPTIVE GREY-BOX FUZZ-TESTING WITH THOMPSON SAMPLING

Sidd Karamcheti, David Rosenberg, Gideon Mann
Bloomberg - CTO Data Science
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 FUZZING & AFL

➤ Grey-Box Mutational Fuzzers

➤ Lightweight instrumentation, to check code path

➤ Mutate existing inputs to generate new test inputs

➤ Fast, efficient, and proven - can find lots of bugs

➤ AFL is best of the bunch!

---

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
GREY-BOX FUZZING & AFL – HEURISTICS

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

Weighted by scores (execution time, program depth)
RANDOM!
Q: How can we do better?

A: *Data-Driven, Adaptive Control!*
GREY-BOX FUZZING & AFL – RELATED WORK

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 $i \in$ range(energy) do
7:     child ← parent
8:     for $j \in$ 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
GREY-BOX FUZZING & AFL – RELATED WORK

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

Neural Byte Sieve for Fuzzing
Rajpal et. al., 2017

FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage
Lemieux and Sen, 2017
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 while

<table>
<thead>
<tr>
<th>Mutation Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitflips</td>
<td>Flip single bit</td>
</tr>
<tr>
<td>Interesting Values</td>
<td>NULL, -1, 0, etc.</td>
</tr>
<tr>
<td>Addition</td>
<td>Add random value</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Subtract random value</td>
</tr>
<tr>
<td>Random Value</td>
<td>Insert random value</td>
</tr>
<tr>
<td>Deletion</td>
<td>Delete from parent</td>
</tr>
<tr>
<td>Cloning</td>
<td>Clone/add from parent</td>
</tr>
<tr>
<td>Overwrite</td>
<td>Replace with random</td>
</tr>
<tr>
<td>Extra Overwrite</td>
<td>Extras: strings scraped</td>
</tr>
<tr>
<td>Extra Insertion</td>
<td>from binary</td>
</tr>
</tbody>
</table>
**MOTIVATING EXAMPLE**

<table>
<thead>
<tr>
<th>Mutation Operation</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitflips</td>
<td>Flip single bit</td>
</tr>
<tr>
<td>Interesting Values</td>
<td>NULL, -1, 0, etc.</td>
</tr>
<tr>
<td>Addition</td>
<td>Add random value</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Subtract random value</td>
</tr>
<tr>
<td>Random Value</td>
<td>Insert random value</td>
</tr>
<tr>
<td>Deletion</td>
<td>Delete from parent</td>
</tr>
<tr>
<td>Cloning</td>
<td>Clone/add from parent</td>
</tr>
<tr>
<td>Overwrite</td>
<td>Replace with random</td>
</tr>
<tr>
<td>Extra Overwrite</td>
<td>Extras: strings scraped from binary</td>
</tr>
<tr>
<td>Extra Insertion</td>
<td></td>
</tr>
</tbody>
</table>

```c
// Runs the ASCII Content Server
int main(void) {
    // Initialize server
    InitializeTree();

    // Respond to commands
    Command command, int more = {}, 1;
    while (more) {
        ReceiveCommand(&command, &more);
        HandleCommand(&command);
    }
    return 0;
}

// Receives and parses an incoming command
int ReceiveCommand(Command *command, int *more) {
    char buffer[64], size_t bytes_received;
    read_until(buffer, ':', sizeof(buffer));

    switch (buffer) {
        case "REQUEST": command->c = REQUEST; break;
        case "QUERY": command->c = QUERY; break;
        case "SEND": command->c = SEND; break;
        case "VISUALIZE": command->c = VISUALIZE; break;
        case "INTERACT": command->c = INTERACT; break;
        default: more = 0; return -1;
    }
    parse_data(command, read_rest(), more);
    return 0;
}
```
Q: How do we identify the right mutators?

A: Learn it!  
*Best indicator of future success is past success*
PICK_MUTATION AS A MULTI-ARMED BANDIT

- Multi-Armed Bandit Problem
  - $k$ "arms" (mutators), each with different probability of paying out (discovering new code path)
  - Starts out unobserved
  - Need to discover the "best" arm (or best distribution over arms) to maximize payout

- Requires a balance between exploration and exploitation
THOMPSON SAMPLING FOR EXPLORATION/EXPLOITATION

➤ Exploration-Exploitation as Bayesian Posterior Estimation

➤ Each mutator has prior \( \pi(\theta) \) - draws parameterize Bernoulli distribution (0/1 reward)

➤ \( \pi(\theta) \propto \theta^{\alpha-1} (1 - \theta)^{\beta-1} \) (Beta-Bernoulli form)

➤ Collect data \( D \) for fixed interval (we used 3 minutes)

➤ Count how many times each mutator was used in generating a "successful" input (S) vs otherwise (F)

➤ Compute Posterior with new info

➤ \( \pi(\theta | D) \propto \theta^{\alpha-1+S} (1 - \theta)^{\beta-1+F} \)

➤ Use posteriors for each arm to obtain mutator distribution
We test our approach on two datasets:

- **DARPA Cyber Grand Challenge Binaries**
  - Set of 200 binaries released by DARPA
  - Each binary has a real "bug" added by a human user
  - We utilize the 150 binaries that read from STDIN

- **LAVA-M Binaries**
  - 4 Binaries from Coreutils
  - Injected with 100s of synthetic bugs
  - If MAGIC_1 < INPUT[10:18] < MAGIC_2: Crash
EXPERIMENTS – BASELINES

➤ We utilize 3 baselines in comparison to our Thompson Sampling approach:

➤ AFL (Vanilla): Same as original algorithm, with extra deterministic step.

➤ FidgetyAFL (Havoc): Original algorithm, implements Böhme et. al Power Schedule for input selection!

➤ Empirical: Test "adaptive" in Thompson Sampling

➤ On 75 of CGC binaries, estimate "empirical" mutator distribution
# RESULTS – CGC BINARIES

- **CGC Binaries (on 75 test programs):**

<table>
<thead>
<tr>
<th></th>
<th>6 hr</th>
<th>12 hr</th>
<th>18 hr</th>
<th>24 hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>0.64 ± 0.03</td>
<td>0.63 ± 0.03</td>
<td>0.63 ± 0.03</td>
<td>0.63 ± 0.03</td>
</tr>
<tr>
<td>FidgetyAFL</td>
<td>0.84 ± 0.02</td>
<td>0.84 ± 0.02</td>
<td>0.85 ± 0.02</td>
<td>0.84 ± 0.02</td>
</tr>
<tr>
<td>Empirical</td>
<td>0.85 ± 0.02</td>
<td>0.86 ± 0.02</td>
<td>0.86 ± 0.02</td>
<td>0.87 ± 0.02</td>
</tr>
<tr>
<td><strong>Thompson</strong></td>
<td><strong>0.91 ± 0.02</strong></td>
<td><strong>0.92 ± 0.02</strong></td>
<td><strong>0.92 ± 0.02</strong></td>
<td><strong>0.93 ± 0.02</strong></td>
</tr>
</tbody>
</table>

**Relative Coverage Statistics (# paths discovered / max)**

<table>
<thead>
<tr>
<th></th>
<th>Crashes</th>
<th>Wins / FidgetyAFL</th>
<th>Wins / All</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>554</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>FidgetyAFL</td>
<td>780</td>
<td>—</td>
<td>14</td>
</tr>
<tr>
<td>Empirical</td>
<td>766</td>
<td>41</td>
<td>5</td>
</tr>
<tr>
<td><strong>Thompson</strong></td>
<td><strong>1336</strong></td>
<td><strong>52</strong></td>
<td><strong>47</strong></td>
</tr>
</tbody>
</table>
## RESULTS – LAVA-M BINARIES

> LAVA-M Binaries:

<table>
<thead>
<tr>
<th></th>
<th>base64</th>
<th>md5sum</th>
<th>uniq</th>
<th>who</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AFL</strong></td>
<td>117 ± 20</td>
<td>55 ± 2</td>
<td>13 ± 0</td>
<td>37 ± 1</td>
</tr>
<tr>
<td><strong>FidgetyAFL</strong></td>
<td>133 ± 10</td>
<td>340 ± 10</td>
<td>87 ± 1</td>
<td>372 ± 36</td>
</tr>
<tr>
<td><strong>Empirical</strong></td>
<td>134 ± 8</td>
<td>406 ± 22</td>
<td>80 ± 1</td>
<td>115 ± 9</td>
</tr>
<tr>
<td><strong>Thompson</strong></td>
<td>144 ± 14</td>
<td>405 ± 2</td>
<td>75 ± 2</td>
<td>106 ± 16</td>
</tr>
</tbody>
</table>

*Number of unique code paths discovered*

<table>
<thead>
<tr>
<th></th>
<th>base64</th>
<th>md5sum</th>
<th>uniq</th>
<th>who</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AFL</strong></td>
<td>15 ± 5</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td><strong>FidgetyAFL</strong></td>
<td>26 ± 9</td>
<td>4 ± 1</td>
<td>1 ± 1</td>
<td>201 ± 56</td>
</tr>
<tr>
<td><strong>Empirical</strong></td>
<td>22 ± 5</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>78 ± 17</td>
</tr>
<tr>
<td><strong>Thompson</strong></td>
<td>31 ± 8</td>
<td>1 ± 1</td>
<td>0 ± 0</td>
<td>106 ± 16</td>
</tr>
</tbody>
</table>

*Number of unique crashes discovered*
RESULTS - GRAPHS

**Discovered Paths over Time for Program ASCII_Content_Server**

- **AFL**
- **FidgetyAFL**
- **Empirical**
- **Thompson**

**Discovered Paths over Time for Program Fortress**

- **AFL**
- **FidgetyAFL**
- **Empirical**
- **Thompson**
FUTURE WORK

➤ Non-Stationary Distributions - need to adapt sampling

➤ Can we get Thompson Sampling to work together with other learned/data-driven heuristics?
  ➤ Well...

<table>
<thead>
<tr>
<th></th>
<th>24 hr</th>
<th>Crashes</th>
<th>Wins / All</th>
</tr>
</thead>
<tbody>
<tr>
<td>FairFuzz</td>
<td>0.88 ± 0.02</td>
<td>734</td>
<td>17</td>
</tr>
<tr>
<td>Thompson</td>
<td>0.95 ± 0.01</td>
<td>1287</td>
<td>49</td>
</tr>
<tr>
<td>FairFuzz + Thompson</td>
<td>0.57 ± 0.03</td>
<td>245</td>
<td>1</td>
</tr>
</tbody>
</table>

➤ Modeling Programs Directly - how inputs behave!
  ➤ Talk to me after about entropy ranking!
SUMMARY

➤ Grey-box Mutational Fuzzing is good, but inefficient
  ➤ Heuristics are non-optimal, can lead to redundant work
➤ Our contribution: improve grey-box fuzzing with data-driven learning!
  ➤ Change distribution over mutators adaptively, via Thompson Sampling - focus on mutators that matter!
  ➤ Results show huge gains over baselines, but not perfect

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

Thank You!