ADAPTIVE GREY-BOX FUZZ-TESTING WITH THOMPSON SAMPLING

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

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 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!

```
// Core Algorithm for American Fuzzy Lop (AFL)
// time: Fixed time window to fuzz (e.g. 24 hours)
// queue: Queue of inputs that exercise new code paths.
while time has not elapsed do
    parent, energy ← pick_input(queue)
    for i ∈ range(energy) do
        child ← parent
        for j ∈ 1 to sample_num_mutations() do
            mutation ← sample_mutation()
            site ← sample_mutation_site()
            child ← apply_mutation(mutation, child, site)
        end for
        path ← execute_path(child, code)
        if (path is new) then queue ← child
    end for
end while
```

GREY-BOX FUZZING & AFL – HEURISTICS

```
// Core Algorithm for American Fuzzy Lop (AFL)
// time: Fixed time window to fuzz (e.g. 24 hours)
// queue: Queue of inputs that exercise new code paths.
while time has not elapsed do
    parent, energy ← pick_input(queue)
    for i ∈ range(energy) do
        child ← parent
        for j ∈ 1 to sample_num_mutations() do
            mutation ← sample_mutation()
            site ← sample_mutation_site()
            child ← apply_mutation(mutation, child, site)
        end for
        path ← execute_path(child, code)
        if (path is new) then queue ← child
    end for
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

```
// Core Algorithm for American Fuzzy Lop (AFL)
// time: Fixed time window to fuzz (e.g. 24 hours)
// queue: Queue of inputs that exercise new code paths.
while time has not elapsed do
    parent, energy ← pick_input(queue)
    for i ∈ range(energy) do
        child ← parent
        for j ∈ 1 to sample_num_mutations() do
            mutation ← sample_mutation()
            site ← sample_mutation_site()
            child ← apply_mutation(mutation, child, site)
        end for
        path ← execute_path(child, code)
        if (path is new) then queue ← child
    end for
end while
```

AFLFast

Coverage-Based Grey-Box Fuzzing

Böhme et. al, 2016

AFLGo

Directed Greybox Fuzzing

Böhme et. al, 2017

GREY-BOX FUZZING & AFL – RELATED WORK

```
// Core Algorithm for American Fuzzy Lop (AFL)
// time: Fixed time window to fuzz (e.g. 24 hours)
// queue: Queue of inputs that exercise new code paths.
while time has not elapsed do
    parent, energy ← pick_input(queue)
    for i ∈ range(energy) do
        child ← parent
        for j ∈ 1 to sample_num_mutations() do
            mutation ← sample_mutation()
            site ← sample_mutation_site()
            child ← apply_mutation(mutation, child, site)
        end for
        path ← execute_path(child, code)
        if (path is new) then queue ← child
    end for
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

GREY-BOX FUZZING & AFL – WHAT'S NEXT?

```
// Core Algorithm for American Fuzzy Lop (AFL)
// time: Fixed time window to fuzz (e.g. 24 hours)
// queue: Queue of inputs that exercise new code paths.
while time has not elapsed do
    parent, energy ← pick_input(queue)
    for i ∈ range(energy) do
        child ← parent
        for j ∈ 1 to sample_num_mutations() do
            mutation ← sample_mutation()          ???
            site ← sample_mutation_site()
            child ← apply_mutation(mutation, child, site)
        end for
        path ← execute_path(child, code)
        if (path is new) then queue ← child
    end for
end while
```

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

MOTIVATING EXAMPLE

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

```
// 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
    - $\pi(\theta \mid D) \propto \theta^{\alpha-1+S} (1-\theta)^{\beta-1+F}$
  - Use posteriors for each arm to obtain mutator distribution

EXPERIMENTS - DARPA CGC & LAVA-M

- 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):

	6 hr	12 hr	18 hr	24 hr
AFL	0.64 ± 0.03	0.63 ± 0.03	0.63 ± 0.03	0.63 ± 0.03
FidgetyAFL	0.84 ± 0.02	0.84 ± 0.02	0.85 ± 0.02	0.84 ± 0.02
Empirical	0.85 ± 0.02	0.86 ± 0.02	0.86 ± 0.02	0.87 ± 0.02
Thompson	**0.91 ± 0.02**	**0.92 ± 0.02**	**0.92 ± 0.02**	**0.93 ± 0.02**

Relative Coverage Statistics (# paths discovered / max)

	Crashes	Wins / FidgetyAFL	Wins / All
AFL	554	18	4
FidgetyAFL	780	—	14
Empirical	766	41	5
Thompson	**1336**	**52**	**47**

Crash Statistics (Unique Paths triggering Crash)

RESULTS - LAVA-M BINARIES

- LAVA-M Binaries:

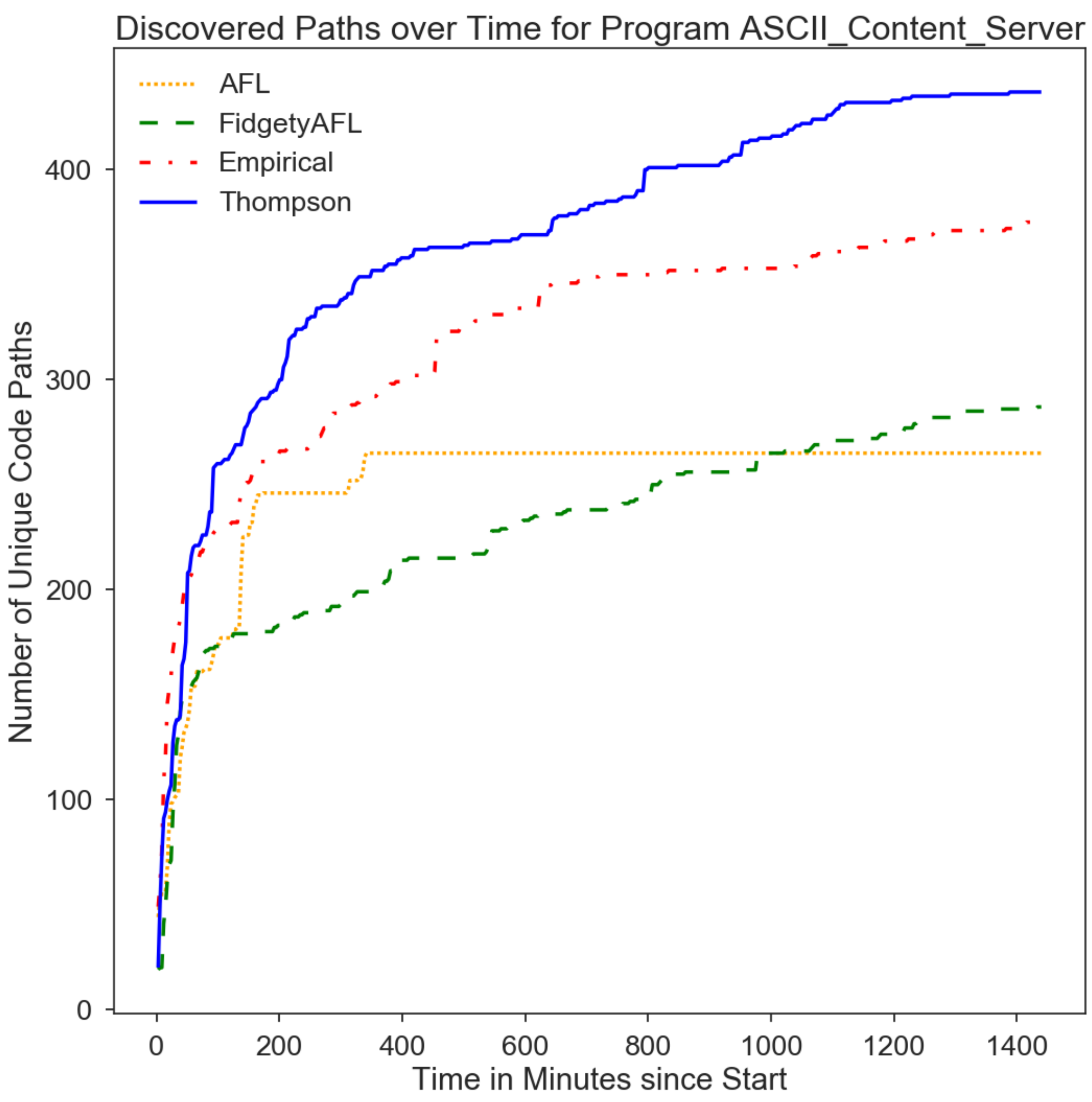
	base64	md5sum	uniq	who
AFL	117 ± 20	55 ± 2	13 ± 0	37 ± 1
FidgetyAFL	133 ± 10	340 ± 10	87 ± 1	372 ± 36
Empirical	134 ± 8	406 ± 22	80 ± 1	115 ± 9
Thompson	144 ± 14	405 ± 2	75 ± 2	106 ± 16

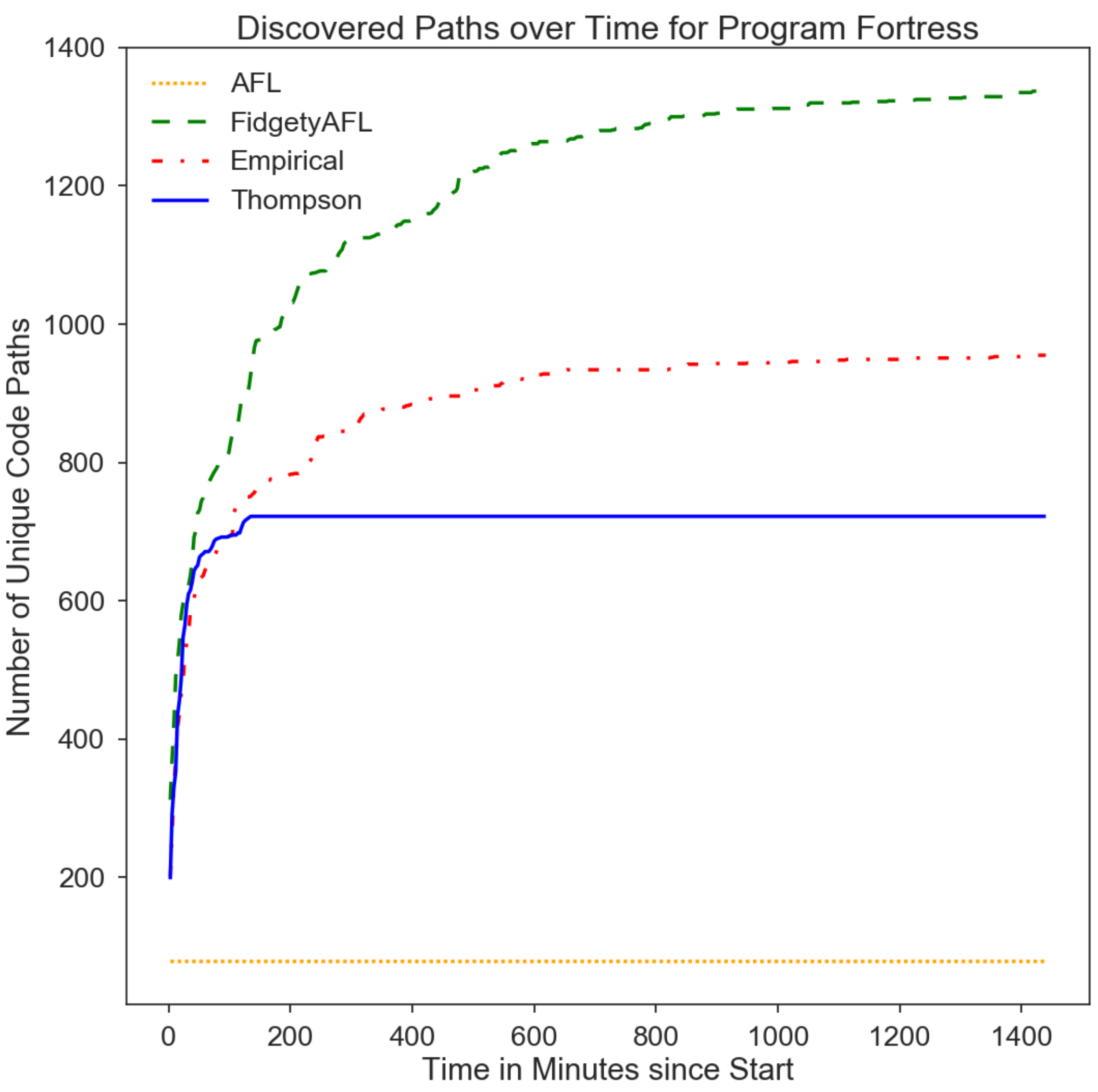
Number of unique code paths discovered

	base64	md5sum	uniq	who
AFL	15 ± 5	0 ± 0	0 ± 0	0 ± 0
FidgetyAFL	26 ± 9	4 ± 1	1 ± 1	201 ± 56
Empirical	22 ± 5	0 ± 0	0 ± 0	78 ± 17
Thompson	31 ± 8	1 ± 1	0 ± 0	106 ± 16

Number of unique crashes discovered

RESULTS - GRAPHS





FUTURE WORK

- Non-Stationary Distributions - need to adapt sampling
- Can we get Thompson Sampling to work together with other learned/data-driven heuristics?
  - Well…

	24 hr	Crashes	Wins / All
FairFuzz	0.88 ± 0.02	734	17
Thompson	**0.95 ± 0.01**	**1287**	**49**
FairFuzz + Thompson	*0.57 ± 0.03*	*245*	*1*

- 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!