Coverage-based Greybox Fuzzing as Markov Chain

Marcel Böhme Van-Thuan Pham Abhik Roychoudhury

School of Computing, National University of Singapore, Singapore
{marcel,thuanpv,abhik}@comp.nus.edu.sg

ABSTRACT

Coverage-based Greybox Fuzzing (CGF) is a random testing approach that requires no program analysis. A new test is generated by slightly mutating a seed input. If the test exercises a new and interesting path, it is added to the set of seeds; otherwise, it is discarded. We observe that most tests exercise the same few "high-frequency" paths and develop strategies to explore significantly more paths with the same number of tests by gravitating towards low-frequency paths.

We explain the challenges and opportunities of CGF using a Markov chain model which specifies the probability that fuzzing the seed that exercises path i generates an input that exercises path j. Each state (i.e., seed) has an *energy* that specifies the number of inputs to be generated from that seed. We show that CGF is considerably more efficient if energy is *inversely proportional to the density of the stationary distribution* and *increases monotonically* every time that seed is chosen. Energy is controlled with a power schedule.

We implemented the exponential schedule by extending AFL. In 24 hours, AFLFast exposes 3 previously unreported CVEs that are not exposed by AFL and exposes 6 previously unreported CVEs 7x faster than AFL. AFLFast produces at least an order of magnitude more unique crashes than AFL.

CCS Concepts:

•Security and privacy→Vulnerability scanners; •Software and its engineering→Software testing and debugging;

1. INTRODUCTION

> *"Ultimately, the key to winning the hearts and minds of practitioners is very simple: you need to show them how the proposed approach finds new, interesting bugs in the software they care about."* – Michal Zalewski [27]

Recently, there has been much debate about the efficiency of symbolic execution-based fuzzers versus more lightweight fuzzers. Symbolic execution is a systematic effort to stress different behaviors and thus considerably more effective. Yet, today most vulnerabilities are exposed by particularly lightweight fuzzers that do not leverage *any* program analysis [27].

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978428

It turns out that even the most effective technique is less efficient than blackbox fuzzing if the time spent generating a test case takes relatively too long [3]. Symbolic execution is very effective because each new test exercises a different path in the program. However, this effectiveness comes at the cost of spending *significant time doing program analysis and constraint solving.* Blackbox fuzzing, on the other hand, does not require any program analysis and generates several orders of magnitude more tests in the same time.

Coverage-based Greybox Fuzzing (CGF) is an attempt to make fuzzing more effective at path exploration *without* sacrificing time for program analysis. CGF uses lightweight (binary) instrumentation to determine a unique identifier for the path that is exercised by an input. New tests are generated by slightly mutating the provided seed inputs (we also call the new tests as *fuzz*). If some fuzz exercises a new and interesting path, the fuzzer retains that input; otherwise, it discards that input. The provided and retained seeds are fuzzed in a continuous loop, contributing even more seeds.

Compared to symbolic execution, CGF does not require program analysis which brings several benefits. There is *no imprecision*, for instance, in the lifting of the control-flow graph from the program binary or the encoding of the path condition as SMT formula. CGF is more *scalable* because the time to generate a test does not increase with the program size. CGF is highly *parallelizable* because the retained seeds represent the only internal state. AFL implements the state-of-the-art of CGF, is behind hundreds of high-impact vulnerability discoveries [21], has been shown to generate valid image files (JPEGs) from an initial seed that is virtually empty [24], and has also been integrated with symbolic execution (which helps where AFL "gets stuck") [19]. Clearly, increasing the efficiency of fuzzers, like AFL, has a real and practical impact on vulnerability detection.

We discuss challenges of existing CGFs and opportunities to boost their efficiency by an order of magnitude. Our key observation is that *most fuzz exercises the same few paths*: Existing CGFs generate too many inputs which stress the same behavior. More efficient CGFs exercise more paths with the same amount of fuzz. For instance, suppose we want to expose vulnerabilities in `libpng`. Fuzzing a *valid* image file, there may be a 90% chance that a mutated variant exercises a path π which rejects invalid image files. Fuzzing an *invalid* image file, there may be a 99.999% chance that a mutated variant exercises the same path π. So, independent of the initial seed image files, an above-average amount of fuzz is bound to exercise that path π which rejects invalid inputs. Informally, we call π a *high-frequency* path.

In this paper, we propose several strategies to focus most of the fuzzing effort on low-frequency paths so as to explore more paths with the same amount of fuzz. The results are very encouraging. Our AFL extension AFLFast discovered 9 vulnerabilities in GNU binutils which are now listed as CVEs in the US National Vulnerability Database. AFLFast exposes 6 CVEs up to 14 times faster than AFL and exposes 3 CVEs that are not exposed by AFL in eight runs of 24 hours. AFLFast reports an order of magnitude more unique crashes than AFL.[1] We will argue that our strategies have no detrimental impact on the effectiveness of AFL. So, given more than 24 hours, AFL is expected to report the same unique crashes and expose the three remaining CVEs.

To explain the remarkable performance gains of AFLFast, we model Coverage-based Greybox Fuzzing (CGF) as Markov chain. The chain specifies the probability p_{ij} that fuzzing the seed exercising path i generates an input exercising path j. We let each state (i.e., seed) have an *energy* that specifies the amount of fuzz that is generated by fuzzing that seed when it is chosen next. For instance, the *minimum energy* required to discover a new and interesting path j by fuzzing the seed which exercises path i is expected to be p_{ij}^{-1}. However, in practice p_{ij} is clearly unknown. The energy of a state is controlled by a pre-defined *power schedule*.

AFL implements a power schedule that assigns an energy that is *constant* in the number of times $s(i)$ the seed has been chosen for fuzzing. Almost every time the seed is chosen, the same number of inputs are generated. In some cases, AFL might assign significantly more than the minimum energy required to discover a new and interesting path; in other cases, AFL might assign not enough energy. In contrast, AFLFast implements a power schedule that assigns an energy that is *exponential* in $s(i)$. When the seed is fuzzed for the first time, very low energy is assigned. Every time the seed is chosen thereafter, exponentially more inputs are generated up to a certain bound. This allows to rapidly approach the minimum energy required to discover a new path.

In fact, AFL implements a power schedule that assigns constantly *high energy*. Often, 80k inputs are generated for each seed which takes about one minute. This addresses the problem of *rapid mixing*: Independent of the initial seed inputs, after a (burn-in) time some paths will always be exercised by significantly more fuzz than others. Assigning a lot of energy to the inital seeds allows to discover many more "neighbors" that exercise *low*-frequency paths. For instance, it makes sense to fuzz a valid image file for the longest time with the objective to generate many more valid image files. It is also a good idea to assign a lot of energy to these neighbors and their neighbors. However, after some time, as more seeds are discovered, many seeds will start to exercise *high*-frequency paths and AFL ends up assigning way too much energy. The chance to generate a valid image file is significantly lower if the latest seed is an invalid image file.

In contrast, AFLFast implements a power schedule that assigns energy that is *inversely proportional* to the density of the stationary distribution. In other words, it assigns low energy to seeds exercising high-frequency paths and high energy to seeds exercising low-frequency paths. We approximate the density of the stationary distribution by counting the number of fuzz $f(i)$ that exercises path i.

[1]AFL reports an input that exercises a new and interesting path *and crashes the program* (i.e., which would otherwise be retained as new seed) as a *unique crash*.

AFL chooses seeds in the order they are added. Once all seeds have been fuzzed, AFL resumes with the first. A new cycle begins. AFLFast effectuates a different *search strategy*. It chooses seeds in the order of their likely progressiveness (while choosing a seed only once per cycle). In the same cycle, AFLFast chooses seeds earlier i) that exercise lower-frequency paths and ii) that have been chosen less often. The search strategy allows to fuzz the best seeds early on. However, independent of the search strategy and given the same power schedule, when a cycle is completed the same seeds will have been fuzzed.

We note that power schedules and search strategies merely impact AFL's *efficiency* (i.e., #paths explored per unit time), *not* its *effectiveness* (i.e., #paths explored in expectation). Since we do not modify the mutation operators[2] that are being used for fuzzing, the probability p_{ij} to discover path j by fuzzing the input exercising path i does not change from AFL to AFLFast. In other words, AFLFast exposes exactly the same vulnerabilities as AFL – only significantly faster.

In summary, we argue that the effectivness of symbolic execution stems from the systematic enumeration of paths in the program. This allows to expose vulnerabilities that hide deep in the program. Unfortunately, most fuzzers trade this systematic path coverage for scalability. However, coverage-based greybox fuzzers maintain some of this effectiveness by retaining fuzz that exercises paths that have previously not been exercised. Each new seed results in progress towards generating even more seeds that exercise even "deeper" paths. Still, even coverage-based fuzzers tend to visit certain paths with high frequency, generating too much fuzz that exercises the same few paths. Our main conceptual contribution is to smartly control the amount of fuzz generated from a seed, thereby veering the search towards paths that are exercised with low frequency, towards paths where vulnerabilities may lurk. Technically, we achieve this enhanced path coverage using power schedules and search strategies that do not require program analysis. Since CGF is highly parallelizable, an efficiency improvement of one order of magnitude for one AFL instance should result in an improvement of about $1 + \log_{10}(N)$ orders of magnitude for N instances.

Specifically, our paper makes the following *contributions*:

- **Markov Chain Model**. We model coverage-based greybox fuzzing as a systematic exploration of the state space of a Markov chain. We provide insight about the machinery that drives AFL, which is arguably *the* most successful vulnerability detection tool to date. We utilize the model to explain the challenges of AFL and the remarkable performance gains of our tool AFLFast.
- **Power Schedules**. We introduce and evaluate several strategies to control the number of inputs generated from a seed, with the objective to exercise a larger number of low-frequency paths in the same time.
- **Search Strategies**. We devise and evaluate several strategies to control the order in which seeds are chosen for fuzzing, with the same objective.
- **Tool**. We publish AFLFast as a fork of AFL. AFLFast was used by Team Codejitsu who came in 2nd *in terms of number of bugs found*[3] at the DARPA Cyber Grand Challenge: **https://github.com/mboehme/aflfast**

[2]AFL's mutation operators include bit flips, boundary value substitution, simple arithmetics & block deletion/insertion.
[3]See red result bar for Galactica at http://bit.do/cgcresults.

2. BACKGROUND

2.1 Coverage-based Greybox Fuzzing

FUZZ – an automated random testing tool was first developed by Miller et al. [13] in early 1990s to understand the reliability of UNIX tools. Since then, fuzzing has evolved substantially, become widely adopted into practice, and exposed serious vulnerabilities in many important software programs [23, 25, 26, 22]. There are three major categories depending on the degree of leverage of internal program structure: *black-box fuzzing* only requires the program to execute [23, 25, 28] while *white-box fuzzing* [5, 11, 8, 9] requires binary lifting and program analysis, for instance, to construct the control-flow graph. *Greybox fuzzing* is situated inbetween and uses only lightweight binary instrumentation to glean some program structure. Without program analysis, greybox fuzzing may be more efficient than whitebox fuzzing. With more information about internal structure, it may be more effective than blackbox fuzzing.

Coverage-based greybox fuzzers (CGF) [22] use lightweight instrumentation to gain coverage information. For instance, AFL's instrumentation captures basic block transitions, along with coarse branch-taken hit counts. A sketch of the code that is injected at each branch point in the program is shown in Listing 1:

```
1 cur_location = <COMPILE_TIME_RANDOM>;
2 shared_mem[cur_location ^ prev_location]++;
3 prev_location = cur_location >> 1;
```

Listing 1: AFL's instrumentation.

The variable `cur_location` identifies the current basic block. Its random identifier is generated at compile time. Variable `shared_mem[]` is a 64 kB shared memory region. Every byte that is set in the array marks a hit for a particular tuple (A, B) in the instrumented code where basic block B is executed after basic block A. The shift operation in Line 3 preserves the directionality [(A, B) versus (B, A)]. A hash computed over the elements in `shared_mem[]` is used as the path identifier.

A CGF uses the coverage information to decide which generated inputs to retain for fuzzing, which input to fuzz next and for how long. Algorithm 1 provides a general overview of the process and is illustrated in the following by means of AFL's implementation. If the CGF is provided with seeds S, they are added to the queue T; otherwise an empty file is generated as a starting point (lines 1–5). The seeds are choosen in a continuous loop until a timeout is reached or the fuzzing is aborted (line 7). AFL classifies a seed as a *favorite* if it is the fastest and smallest input for any of the control-flow edges it exercises. AFL's implementation of CHOOSENEXT mostly ignores non-favorite seeds.

For each seed input t, the CGF determines the number of inputs that are generated by fuzzing t (i.e., #fuzz for t; line 8). AFL's implementation of ASSIGNENERGY uses the execution time, block transition coverage, and creation time of t. Then, the fuzzer generates p new inputs by mutating t according to defined mutation operators (line 10). AFL's implementation of MUTATE_INPUT uses bit flips, simple arithmetics, boundary values, and block deletion and insertion strategies to generate new inputs.[4]

[4]https://lcamtuf.blogspot.sg/2014/08/binary-fuzzing-strategies-what-works.html

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S
1: $T_{\text{✗}} = \emptyset$
2: $T = S$
3: **if** $T = \emptyset$ **then**
4: add empty file to T
5: **end if**
6: **repeat**
7: t = CHOOSENEXT(T)
8: p = ASSIGNENERGY(t)
9: **for** i from 1 to p **do**
10: t' = MUTATE_INPUT(t)
11: **if** t' crashes **then**
12: add t' to $T_{\text{✗}}$
13: **else if** ISINTERESTING(t') **then**
14: add t' to T
15: **end if**
16: **end for**
17: **until** *timeout* reached or *abort*-signal
Output: Crashing Inputs $T_{\text{✗}}$

If the generated input t' is considered to be "interesting", it is added to the circular queue (line 14). AFL's implementation of ISINTERESTING returns true depending on the number of times the basic block transitions, that are executed by t', have been executed by other seeds in the queue. More specifically, t' *is interesting* if t' executes a path where transition b is exercised n times and for all other seeds $t'' \in T$ that exercise b for m times, we have that $\lfloor \log_2 n \rfloor \neq \lfloor \log_2 m \rfloor$ where $\lfloor \cdot \rfloor$ is the floor function. AFL uses this "bucketing" to address path explosion [19]. Intuitively, AFL retains inputs as new seeds that execute a new block transition or a path where a block transition is exercised twice when it is normally exercised only once. At the same time, AFL discards inputs that execute a path where some transition is exercised 102 times when it has previously been exercised 101 times. If the generated input t' crashes the program, it is added to the set $T_{\text{✗}}$ of crashing inputs (line 12). A crashing input that is also interesting is marked as *unique crash*.

Binary instrumentation. AFL supports both, source code instrumentation and binary instrumentation via QEMU [1]. While QEMU does the instrumentation during interpretation at runtime, AFLDynInst [20] injects the instrumentation shown in Listing 1 directly into the binary.

Modifications. Our changes of AFL concern *only* the functions CHOOSENEXT which implements the search strategy and ASSIGNENERGY which implements the power schedules.

2.2 Markov Chain

A Markov chain is a stochastic process that transitions from one state to another [14]. At any time, the chain can be in only one state. The set of all states is called the chain's *state space*. The process transitions from one state to another with a certain probability that is called the *transition probability*. This probability depends only upon the current state rather than upon the path to the present state.

More formally, a *Markov chain* is a sequence of random variables $\{X_0, X_1, \ldots, X_n\}$ where X_i describes the state of the process at time i. Given a set of states $S = \{1, 2, \ldots, N\}$ for some $N \in \mathbb{N}$, the value of the random variables X_i are taken from S. The probability that the Markov chain starts out in state i is given by the initial distribution $\mathbb{P}(X_0 = i)$.

The *probability matrix* $\boldsymbol{P} = (p_{ij})$ specifies the transition rules. If $|S| = N$, then $\boldsymbol{P}$ is a $N \times N$ stochastic matrix where each entry is non-negative and the sum of each row is 1. The conditional probability p_{ij} defines the probability that the chain transitions to state j at time $t+1$, given that it is in state i at time t,

$$p_{ij} = \mathbb{P}(X_{t+1} = j \mid X_t = i)$$

A Markov chain is called *time-homogeneous* if the probability matrix (p_{ij}) does not depend on the time n. In other words, every time the chain is in state i, the probability of jumping to state j is the same.

If a Markov chain is time homogeneous, then the vector $\boldsymbol{\pi}$ is called a *stationary distribution* of the Markov chain if for all $j \in S$ it satisfies

$$0 \leq \pi_j \leq 1$$
$$1 = \sum_{i \in S} \pi_i$$
$$\pi_j = \sum_{i \in S} \pi_i p_{ij}$$

Informally, a Markov chain $\{X_0, X_1, \ldots, X_n\}$ is called *rapidly mixing* if X_n is "close" to the stationary distribution for a sufficiently low number of steps n. In other words, rapidly mixing Markov chains approach the stationary distribution within a reasonable time – independent of the initial state.

Random walkers sample the distribution that is described by a Markov chain. A walker starts at a state according to the initial distribution and transitions from one state to the next according to the transition probabilities. The state at which the walker arrives after n steps is considered a sample of the distribution. There may be an ensemble of walkers that move around randomly.

For instance, the crawling of web pages can be modelled as Markov chain. Pages are the states while the links are the transitions. Given page i with q_i links where one link goes to page j, the probability p_{ij} that a random surfer reaches j from i in one click is $p_{ij} = 1/q_i$. A crawler, like Google, seeks to index the important pages of the internet. Brin and Page [4] developed an algorithm, called PageRank that assigns an importance score to each page. Intuitively, the PageRank value of a page measures the chance that a random surfer will land on that page after a sequence of clicks. More formally, the PageRank approximates the *density of the stationary distribution* of the Markov chain where important pages are located in high-density regions.

3. MARKOV CHAIN MODEL

In this paper, we model the probability that fuzzing a seed which exercises program path i generates a seed which exercises path j as transition probability p_{ij} in a Markov chain. This allows us to discuss the *objective* of greybox fuzzing as the efficient exploration of the chain's state space and to explain the *challenges and opportunities* of CGF and of AFL specifically. We argue that a coverage-based greybox fuzzer exercises more distinct paths per unit time if it does focus on inputs in low-density regions of the Markov chain. Hence, we devise several strategies to bias the traversal towards visiting more states in low-density regions and less states in high-density regions of the stationary distribution. Before discussing these strategies, we introduce the Markov chain model.

3.1 Coverage-based Fuzzing as Markov Chain

*Time-**in**homogeneous model.* Suppose, after providing the fuzzer with an initial seed input t_0 that exercises path 0, the fuzzer immediately explores path $i+1$ by randomly mutating the previous input t_i which exercises path i. Every input that is generated is directly chosen as next seed. The sequence of paths that the fuzzer exercises is described by a Markov chain. The transition probability p_{ij} is defined as the probability to generate an input that exercises path j by randomly mutating the previous input t_i that exercises path i. Clearly, this Markov chain is not time-homogeneous. The transition probability p_{ij} depends on the path in the Markov chain by which the state i was reached. Say, a different input t_i' is fuzzed that also exercises path i, the probability p_{ij} to generate an input that exercises path j might be very different. While this is still a Markov chain, it is not time-homogeneous. The analysis is difficult and the existence of a stationary distribution is not guaranteed.

Time-homogeneous model. A stationary distribution does exist for the following model of coverage-based fuzzing. The *state space* of the Markov chain is defined by the discovered paths and their immediate neighbors. Given seeds T, let S^+ be the set of (discovered) paths that are exercised by T and S^- be the set of (undiscovered) paths that are exercised by inputs generated by randomly mutating any $t \in T$.[5] Then the set of states S of the Markov chain is given as

$$S = S^+ \cup S^-$$

The *probability matrix* $P = (p_{ij})$ of the Markov chain is defined as follows. If path i is a discovered path exercised by $t_i \in T$ (i.e., $i \in S^+$), then p_{ij} is the probability that randomly mutating seed t_i generates an input that exercises the path j. Else if path i is an undiscovered path that is not exercised by some $t \in T$ (i.e., $i \in S^-$), then $p_{ii} = 1 - \sum_{t_j \in T} p_{ji}$ and $p_{ij} = p_{ji}$ for all $t_j \in T$. In other words, without loss of generality we make the following two assumptions. We assume that generating an input that exercises path j from (undiscovered) seed t_i is as likely as generating from seed t_j an input that exercises (undiscovered) path i. We also assume that $i \in S^-$ has no other undiscovered neighbors.

The *stationary distribution* $\boldsymbol{\pi}$ of the Markov chain gives the probability that a random walker that takes N steps spends roughly $N\pi_i$ time periods in state i. In other words, the proportion of time spent in state i converges to π_i as N goes to infinity. We call a *high-density region* of $\boldsymbol{\pi}$ a neighborhood of paths I where $\mu_{i \in I}(\pi_i) > \mu_{t_j \in T}(\pi_j)$ and μ is the arithmetic mean. Similarly, we call a *low-density region* of $\boldsymbol{\pi}$ a neighborhood of paths I where $\mu_{i \in I}(\pi_i) < \mu_{t_j \in T}(\pi_j)$. It is not difficult to see that a greybox fuzzer is more likely to exercise paths in a high-density region of $\boldsymbol{\pi}$ than in a low-density region. Note that we get a new Markov chain once an undiscovered path $i \in S^-$ is discovered.

Energy & Power Schedules. We let each state $s \in S^+$ have an energy. The *energy* of state i determines the number of inputs that should be generated by fuzzing the seed t_i when t_i is next chosen from the queue T. The energy of a state is controlled by a pre-defined *power schedule*. Note that energy is a local property specific to a state (unlike temperature in simmulated annealing). In Algorithm 1, the power schedule is implemented by the function assignEnergy.

[5] An input t_i is randomly mutated using `mutate_input` on t_i in Algorithm 1.

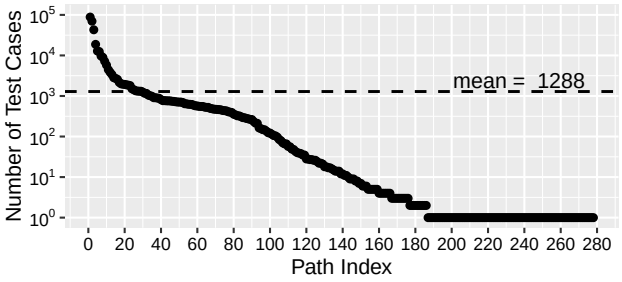

Figure 1: #Fuzz exercising a path (on a log-scale) after running AFL for 10 minutes on the `nm`-tool.

Long tails. In our experiments, we observe several notable properties of the Markov chain model of coverage-based greybox fuzzing. For one, the stationary distribution has a large number of very-low-density regions and a small number of very-high-density regions. As shown in Figure 1, 30% of the paths are exercised by just a single generated test input while 10% of the paths are exercised by $1k$ to $100k$ generated test inputs. In other words, most inputs exercise a few high-frequency paths. Often, these inputs are invalid while the few inputs exercising the low-frequency paths are valid and interesting. Basically, almost each valid input would exercise different behavior. Hence, in this paper we devise strategies to explore such low-density regions more efficiently.

Rapid mixing. Moreover, such Markov chains are mostly rapidly mixing. Given our exploration objective, this is most unfortunate. It takes only a few transitions to "forget" the initial state and arrive in a high-density region that is visited by most walkers. After a few transitions, the probability that the current state corresponds to a *high*-frequency path is high, no matter whether the walker started with an initial seed that exercises a low-frequency path or not, or whether the walker started with a valid or an invalid input.

Benefits. The Markov chain model of coverage-based greybox fuzzing has several benefits. For example, it opens fuzzing for the efficient approximation of numerical program properties, such as the worst-case or average execution time or energy consumption. There exist several Markov Chain Monte Carlo (MCMC) methods, like Simulated Annealing [12] that offer guarantees on the convergence to the actual value. In the context of vulnerabilty research, the Markov chain model allows to explain the challenges and opportunities of existing coverage-based fuzzers, such as AFL.

3.2 Running Example

On a high level, we model the probability that fuzzing a test input $t \in T$ which exercises some path i generates an input which exercises path j as transition probabilities p_{ij} in a Markov chain. We illustrate this model using the simple program in Listing 2 which takes as input a 4-character word and crashes for the input "bad!".

```
1 void crashme (char* s) {
2   if (s[0] == 'b')
3     if (s[1] == 'a')
4       if (s[2] == 'd')
5         if (s[3] == '!')
6           abort();
7 }
```

Listing 2: Motivating example.

The program has five execution paths. Path 0 (`****`) is executed by all strings that do not start with the letter 'b'. Path 1 (`b***`) is executed by all strings starting with "b" that do not continue with the letter 'a'. Path 2 (`ba**`) is executed by all strings starting with "ba" that do not continue with the letter 'd'. Path 3 (`bad*`) is executed by all strings starting with "bad" that do not continue with the letter '!'. Finally, Path 4 is executed only by the input "bad!".

Now, let us specify the implementation of MUTATE_INPUT (MI) in Algorithm 1 to modify a seed input $s = \langle c_0, c_1, c_2, c_3 \rangle$ to generate new inputs. MI chooses with equal probability a character c from s and substitutes it by a character that is randomly chosen from the set of 2^8 ASCII characters. For example, the word "bill" exercises Path 1. With probability 1/4, MI chooses the second character c_1 and with probability $1/2^8$ it chooses the letter 'a' for the substitution. With a total probability of 2^{10}, MI generates the word "ball" from "bill" as the next test input which exercises Path 2.

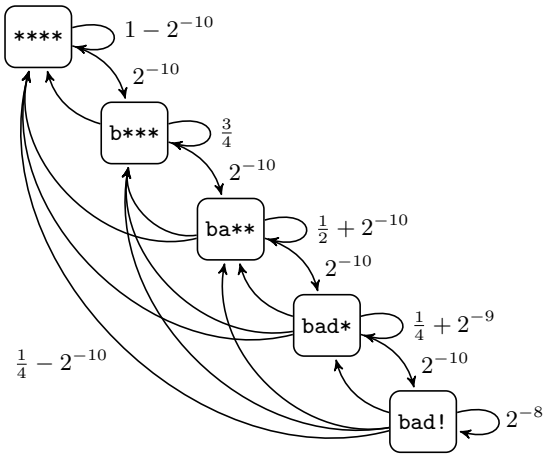

Figure 2: Markov chain for motivating example

Figure 2 represents the simplified transition matrix p_{ij} as a state diagram.[6] For example, if the current input is the word "bill", the Markov Chain is in the state `b***`. The likelihood to transition to the state `ba**` is 2^{-10} as explained earlier. In other words, on average it takes $2^{10} = 1024$ executions of MI on the word "bill" to exercise Path 3 and reach state `ba**`. Given the word "bill", the likelihood to transition to the same state `b***` is 0.75 because MI may choose the first letter and 'b' as substitute or the second letter and any letter except 'a' as substitute with a total probability of 0.25 and it may choose the third or fourth letter with a total probability of 0.5. The probability to transition to state `****` is $\left(1/4 - 2^{-10}\right)$ because MI may choose the first of four letters and substitute it with any letter except 'b'.

Notice that there is a very high probability density in state `****`. Most 4-character words do not start with 'b' such that the initial distribution is heavily biased towards that state. The random walker can transition to the next state only with probability 2^{-10}, stays in `b***` with probability 3/4 and comes back to the state `****` with the approximate probability 1/4. Many inputs will be generated until the walker reaches the state `bad!`.

[6]For simplicity, we ignore some low probability transitions, e.g., from state `****` to state `bad!`.

3.3 Challenges of Coverage-based Fuzzers

A *coverage-based greybox fuzzer* is an ensemble of random walkers in the Markov chain. There is one walker for each seed $t \in T$. The *objective* is to discover an interesting path $s \in S^-$ that is not exercised by any $t \in T$ while generating a minimal number of inputs. Conceptually, all walkers can move simultaneously. Technically, resources are limited and we need to choose which walker can move and how often. In a sequential setting, the fuzzer chooses the next input to fuzz $t \in T$ according to CHOOSENEXT and generates as many inputs as determined by $p = \text{ASSIGNENERGY}(t)$ in Algorithm 1. Usually, $p < M$ where $M \in \mathbb{N}$ gives an upper bound on the number of generated inputs. In AFL, $M \approx 160k$.

More Energy Than Needed. AFL implements a schedule that assigns energy that is *constant* in the number of times the corresponding seed has been chosen from the queue. Let X_{ij} be the random variable that describes the *minimum energy* that should be assigned to state $i \in S^+$ so that the fuzzer discovers the new state $j \in S^-$ where $p_{ij} > 0$. Then,

$$\mathbb{E}[X_{ij}] = \frac{1}{p_{ij}}$$

Now, AFL's constant schedule might assign significantly more or significantly less energy than is actually required.

Example. Let AFL's power schedule assign an energy of $p(i) = 2^{16} = 64k$ to a state i every time t_i is chosen. Since most 4-character words do not start with 'b', the first input t_0 likely exercises Path 0. After 2^{16} inputs have been generated by fuzzing t_0, several inputs are expected to begin with the letter 'b'. One input that exercises Path 1 is retained as seed t_1. After another 2^{16} inputs have been generated by fuzzing t_1, at least one input is expected to exercise Path 2 and is retained as t_2. Figure 3 shows how the procedure continues. After a total of $256k$ inputs were generated from the four seeds that were retained for each path, the crashing input is found. A more efficient fuzzer would need to generate no more than $\mathbb{E}[X_{01}]+\mathbb{E}[X_{12}]+\mathbb{E}[X_{23}]+\mathbb{E}[X_{34}] = 4 \cdot 2^{10} = 4k$ inputs to expose the same vulnerability.

#Total Tests	State	Explored States
1	`****`	`****`
$2^{16}+1$	`b***`	`****`, `b***`
$2 \cdot 2^{16}+1$	`ba**`	`****`, `b***`, `ba**`
$3 \cdot 2^{16}+1$	`bad*`	`****`, `b***`, `ba**`, `bad*`
$4 \cdot 2^{16}+1$	`bad!`	`****`, `b***`, `ba**`, `bad*`, `bad!`

Figure 3: The crash is found after $2^{18} = 256k$ inputs were generated by fuzzing when $p = 2^{16}$ is constant.

Excessive Energy for High-Density Regions. AFL's power schedule also assigns constantly *high energy*: Fuzzing a seed input often takes about a minute on our machine. This addresses the problem of *rapid mixing*. Initial seeds are often provided such that they exercise interesting paths in a low-density region in the stationary distribution of the Markov chain. Assigning high energy to the inital seeds and the seeds in the immediate neighborhood allows to discover many more neighbors in the same low-density region. However, as the retained inputs exercise paths in high-density regions – and there is a natural tendency – too much energy is assigned to these states. By definition, the higher the density of the stationary distribution of the Markov chain for the given state i, the higher the proportion of inputs generated by fuzzing t_i that will exercise high-frequency paths.

	State	`****`	`b***`	`ba**`	`bad*`	`bad!`
1	`ba**`	$1 \cdot 2^7$	$1 \cdot 2^7$	$2 \cdot 2^7$	0	0
2	`****`	$5 \cdot 2^7$	$1 \cdot 2^7$	$2 \cdot 2^7$	0	0
3	`b***`	$6 \cdot 2^7$	$4 \cdot 2^7$	$2 \cdot 2^7$	0	0
4	`ba**`	$7 \cdot 2^7$	$5 \cdot 2^7$	$4 \cdot 2^7$	1	0
5	`****`	$11 \cdot 2^7$	$5 \cdot 2^7$	$4 \cdot 2^7$	1	0
6	`b***`	$12 \cdot 2^7$	$8 \cdot 2^7$	$4 \cdot 2^7$	1	0
7	`bad*`	$13 \cdot 2^7$	$9 \cdot 2^7$	$5 \cdot 2^7$	$1 \cdot 2^7$	0
8	`ba**`	$14 \cdot 2^7$	$10 \cdot 2^7$	$7 \cdot 2^7$	$1 \cdot 2^7$	0
9	`****`	$18 \cdot 2^7$	$10 \cdot 2^7$	$7 \cdot 2^7$	$1 \cdot 2^7$	0
10	`b***`	$19 \cdot 2^7$	$13 \cdot 2^7$	$7 \cdot 2^7$	$1 \cdot 2^7$	0
11	`bad*`	$20 \cdot 2^7$	$14 \cdot 2^7$	$8 \cdot 2^7$	$2 \cdot 2^7$	**1**

Figure 4: Total #fuzz exercising the corresponding path when fuzzing the given state. Too much energy assigned to state `****` and not enough to state `bad*` once it is discovered. Lines indicate new cycles.

Example. Let the initial seed input be the word `ball` and let AFL's power schedule assign an energy of $p(i) = 2^9 = 512$ to a state i every time t_i is chosen. This allows us to discuss the case where the next state is not found in a single fuzzing iteration and several cycles through the circular queue might be required. Recall that AFL chooses the seeds in the order they are added. Figure 4 elaborates the example. After fuzzing the initial seed input for 2^9 times, two new seeds are discovered. About one quarter of the fuzz (i.e., 2^7 inputs) exercises paths `****` and `b***`, respectively (see Fig. 2 and Fig. 4, Row 1). Fuzzing the first discovered seed (Row 2), all fuzz exercises the same path. Fuzzing the second discovered seed (Row 3), a quarter of the fuzz exercises path `****` and three quarters exercises path `b***`. Since no new seeds are discovered, a new cycle begins with the initial seed (Row 4). This procedure continues until the vulnerability is exposed (Row 11). In each row we see that most fuzz exercises path `****`. Evidently, the fuzzer spends way too much time exercising this high-frequency path. The same time would be better spent fuzzing the seed exercising the low-frequency path `bad*`.

In summary, two *challenges* of existing coverage-based greybox fuzzers are: Their power schedules

1. may assign *more energy than is required* in expectation to discover a new and interesting path and
2. may assign *too much energy* to states in *high-density regions* of the chain's stationary distribution and not enough energy to states in low-density regions.

4. BOOSTING GREYBOX FUZZING

A more efficient coverage-based greybox fuzzer discovers an undiscovered state in a *low-density region* while assigning *the least amount of total energy*. More specifically,

1. **Search Strategy**. The fuzzer chooses $i \in S^+$ such that $\exists j \in S^-$ where π_j is low and $\mathbb{E}[X_{ij}]$ is minimal.
2. **Power Schedule**. The fuzzer assigns the energy $p(i) = \mathbb{E}[X_{ij}]$ to the chosen state i in order to limit the fuzzing time to the minimum that is required to be expected to discover a path in a low-density region.

In this paper, we propose *monotonous* power schedules that first assign low energy which monotonously increases every time the corresponding seed is chosen from the queue. This allows to rapidly approach $\mathbb{E}[X_{ij}]$. Moreover, our power schedules assign energy that is *inversely proportional* to the density of the stationary distribution of the Markov chain.

Intuitively, as soon as a new path is discovered, we want to swiftly explore its general neighborhood expending only low energy. This allows us to get a first estimate of whether i lives in a high-density region. Every time i is chosen thereafter, it is assigned more energy. Intuitively, after the neighborhood is explored and it is established that i lives in a low-density region, the fuzzer can invest significantly more energy trying to find paths in the low-density neighborhood of i.

We also propose and evaluate search strategies that are aimed at the fuzzer expending most energy for paths in low-density regions. For instance, to establish whether a state is in a low-density region, we prioritize such $t \in T$ that have been chosen from the circular queue least often and such t that exercise paths that have least often been exercised by other generated test inputs.

4.1 Power Schedules

A power schedule regulates the energy $p(i)$ of a state. More specifically, a power schedule decides how many inputs are generated by fuzzing the seed $t_i \in T$ which exercises path i when t_i is selected next. In general, $p(i)$ is a function of a) the number of times $s(i)$ that t_i has previously been choosen from the queue T and b) the number of generated inputs $f(i)$ that exercise i. In fact, $f(i)$ serves as approximation of the distribution's density. We discuss and evaluate several power schedules.

The **exploitation-based constant schedule** (EXPLOIT) is implemented by most greybox fuzzers. After some burn-in, the assigned energy is fairly constant every time $s(i)$ that t_i is being chosen from the circular queue. The energy $p(i)$ for state i is computed as

$$p(i) = \alpha(i) \qquad \text{e.g., for AFL} \tag{1}$$

where $\alpha(i)$ is the CGF's present implem. of `assignEnergy` in Algorithm 1 and remains constant as $s(i)$ or $f(i)$ varies. For instance, AFL computes $\alpha(i)$ depending on the execution time, block transition coverage, and creation time of t_i. The example in Figure 3 is derived using a constant schedule.

The **exploration-based constant schedule** (EXPLORE) is a schedule that assigns constant but also fairly low energy. The energy $p(i)$ for state i is computed as

$$p(i) = \frac{\alpha(i)}{\beta} \tag{2}$$

where $\alpha(i)/\beta$ maintaints the fuzzer's original judgement $\alpha(i)$ of the quality of t_i and where $\beta > 1$ is a constant.

Cut-Off Exponential (COE) is an exponential schedule that prevents high-frequency paths to be fuzzed until they become low-frequency paths. The COE increases the fuzzing time of t_i exponentially each time $s(i)$ that t_i is chosen from the circular queue. The energy $p(i)$ is computed as

$$p(i) = \begin{cases} 0 & \text{if } f(i) > \mu \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases} \tag{3}$$

where $\alpha(i)$ maintaints the fuzzer's original judgement and $\beta > 1$ is a constant that puts the fuzzer in exploration mode for t_i that have only recently been discovered (i.e., $s(i)$ is low), and where μ is the mean number of fuzz exercising a discovered path

$$\mu = \frac{\sum_{i \in S^+} f(i)}{|S^+|}$$

where S^+ is the set of discovered paths. Intuitively, high-frequency paths where $f(i) > \mu$ that receive a lot of fuzz even from fuzzing other seeds are considered low-priority and not fuzzed at all until they are below the mean again. The constant M provides an upper bound on the number of inputs that are generated per fuzzing iteration.

#Tests	State	Explored States
1	`****`	`****`
2^{10}	`b***`	`****`, `b***`
$2 \cdot 2^{10}$	`ba**`	`****`, `b***`, `ba**`
$3 \cdot 2^{10}$	`bad*`	`****`, `b***`, `ba**`, `bad*`
$4 \cdot 2^{10}$	`bad!`	`****`, `b***`, `ba**`, `bad*`, `bad!`

Figure 5: The crash is found after $2^{12} = 4k$ inputs were generated by fuzzing with a power schedule.

Example. Figure 5 depicts the states that a greybox fuzzer explores with the COE power schedule with $\alpha(i)/\beta = 1$. The first test input is chosen at random from the program's input space. Since most 4-character words do not start with 'b', the first input t_0 likely exercises path 0 which corresponds to state `****`. The first time that t_0 is fuzzed, $s(0) = 0$ and $f(0) = \mu = 1$ so that $\alpha(0) = 2^0$. Next time, $s(0) = 1$ and $f(0) = \mu = 2$ so that $\alpha(0) = 2^1$. When $s(0) = 9$ and $\alpha(0) = 2^9$, 2^{10} test inputs will be generated so that one generated test input t_1 is expected to start with the letter 'b' and the state `b***` is discovered (see Fig. 2). Now, the newly discovered state is assigned low energy $\alpha(1) = 2^0$. However, $f(0) > \mu$ so that soley t_1 will be fuzzed in a similar fashion as t_0 until $s(1) = 9$, $\alpha(1) = 2^9$ and 2^{10} test inputs have been generated by fuzzing t_1. Again, one test input is expected to start with "ba" and the state `ba**` is discovered. Table 5 shows how the procedure continues. After $4k$ test inputs were generated from the four inputs that were retained for each path, the crashing input is found. The random generation of the same string would require *five orders of magnitude* more inputs on average ($4 \cdot 10^6 k$ random inputs) while the constant schedule in Figure 3 would require *one order of magnitude* more test inputs on average ($256k$).

The **exponential schedule** (FAST) is an extension of COE. Instead of not fuzzing t_i at all if $f(i) > \mu$, the power schedule induces to fuzz t_i inversely proportional to the amount of fuzz $f(i)$ that exercises path i. The energy $p(i)$ that this schedule assigns to state i is computed as

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right) \tag{4}$$

Intuitively, $f(i)$ in the denominator allows to exploit t_i that have not received a high number of fuzz in the past and is thus more likely to be in a low-density region. The exponential increase with $s(i)$ allows more and more energy for paths were we are more and more confident that they live in a low-density region.

The **linear schedule** (LINEAR) increases the energy of a state i in a linear manner w.r.t. the number of times $s(i)$ that t_i has been chosen from T, yet is also inversely proportional to the amount of fuzz $f(i)$ that exercises path i.

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right) \tag{5}$$

The **quadratic schedule** (QUAD) increases the energy of a state i in a quadratic manner w.r.t. the number of times $s(i)$ that t_i has been chosen from T, yet is also proportional to the amount of fuzz $f(i)$ that exercises path i. The energy $p(i)$ for state i is computed as

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right) \quad (6)$$

4.2 Search Strategies

While a power schedule regulates the time spent fuzzing a seed, a search strategy decide which seed is chosen next. The decision is purely based on the number the number of times a seed has been fuzzed before and the amount of fuzz exercising the same path as the seed. An efficient coverage-based greybox fuzzer prioritizes inputs that have not been fuzzed very often and inputs that exercise low-frequency paths.

Prioritize small *s(i)*. This strategy chooses $t_i \in T$ such that the number of times $s(i)$ that t_i has been fuzzed is minimal. However, the fuzzer may still decide to skip the choosen test input, for instance if it is not a designated favourite. In that case, the search strategy is applied again until the fuzzer does not skip the input. Effectively, the queue is reordered using the search strategy. Intuitively, the fuzzer can establish early whether or not path i is a low-frequency path and whether it should invest more energy into fuzzing t_i.

Prioritize small *f(i)*. This strategy chooses $t_i \in T$ such that the number $f(i)$ of generated inputs that exercise path i is minimal. The fuzzer may skip the chosen test input, for instance if it is not a designated favourite, until finally an input is chosen according to the search strategy and accepted for fuzzing. Intuitively, fuzzing an input that exercises a low-frequency path might generate more inputs exercising low-frequency paths.

4.3 Implementation of AFLFast

AFL is a coverage-based greybox fuzzer that collects information on the basic block transitions that are exercised by an input. AFL's binary instrumentation is discussed in Section. 2.1. In our experiments, we extended version `1.94b`. AFL implements certain strategies to select "interesting" inputs from the fuzz to add to the queue. We did not change this functionality. AFL addresses path explosion by "bucketing" – the grouping of paths according to the number of times all executed basic block transitions are exercised. We did not change this functionality either. All changes were made to assignEnergy and chooseNext in Algorithm 1.

Changes for Power Schedule. We changed the computation of the amount of fuzz $p(i)$ that is generated for an input t_i. Firstly, AFL computes $p(i)$ depending on execution time, transition coverage, and creation time of t_i. Essentially, if it executes more quickly, covers more, and is generated later, then the number of fuzz is greater. We maintain this evaluation in the various power schedules discussed above. Secondly, AFL executes the deterministic stage the first time t_i is fuzzed. Since our power schedules assign significantly less energy for the first stage, our extension executes the deterministic stage later when the assigned energy is equal to the energy spent by deterministic fuzzing. Lastly, AFL might initially compute a low value for $p(i)$ and then dynamically increase $p(i)$ in the same run if "interesting" inputs are generated. Since our implementation controls $p(i)$ via a power schedule, we disabled this dynamic increase for AFLFast.

Changes for Search Strategy. We changed the order in which AFL chooses the inputs from the queue and how AFL designates "favourite" inputs that are effectively exclusively chosen from the queue. Firstly, for all executed basic block transitions b, AFL chooses as favourite the fastest and smallest inputs executing b. AFLFast first chooses the input exercising b with the smallest number of time $s(i)$ that it has been chosen from the queue, and if there are several, then the input that exercises a path exercised by the least amount of fuzz $f(i)$, and if there are still several, then the fastest and smallest input. Secondly, AFL chooses the next favourite input which follows the current input in the queue. AFLFast chooses the next favourite input with the smallest number of time $s(i)$ that it has been chosen from the queue and if there are several, it chooses that which exercises a path exercised by the least amount of fuzz $f(i)$.

5. EVALUATION

5.1 Vulnerabilities

We chose GNU binutils as subject because it is non-trivial and widely used for the analysis of program binaries. It consists of several tools including `nm`, `objdump`, `strings`, `size`, and `c++filt`. We zoom into some results by discussing the results for `nm` in more detail.[7] Binutils is a difficult subject because the fuzzer needs to generate some approximation of a program binary in order to exercise interesting behaviors of the programs. We found a large number of serious vulnerabilities and several bugs (listed in Table 1).

Table 1: CVE-IDs and Exploitation Type

Vulnerability	Type
CVE-2016-2226	Exploitable Buffer Overflow
CVE-2016-4487	Invalid Write due to a Use-After-Free
CVE-2016-4488	Invalid Write due to a Use-After-Free
CVE-2016-4489	Invalid Write due to Integer Overflow
CVE-2016-4490	Write Access Violation
CVE-2016-4491	Various Stack Corruptions
CVE-2016-4492	Write Access Violation
CVE-2016-4493	Write Access Violation
CVE-2016-6131	Stack Corruption
Bug 1	Buffer Overflow (Invalid Read)
Bug 2	Buffer Overflow (Invalid Read)
Bug 3	Buffer Overflow (Invalid Read)

All vulnerabilities were previously unreported and rated as medium security risk. We informed the maintainers, submitted patches, and informed the security community via the *ossecurity* mailing list.[8] Mitre assigned *nine (9) CVEs*. At the time of writing, all but one patches have been accepted while one is still under review. These vulnerabilities affect most available binary analysis tools including valgrind, gdb, binutils, gcov and other libbfd-based tools. An attacker might modify a program binary such that it executes malicious code upon *analysis*, e.g., an analysis to identify whether the binary is malicious in the first place or during the attempt of reverse-engineering the binary.

Measure of #paths. AFL maintains a unique path indentifier `cksum` for each input in the queue that is computed as a hash over the shared memory region that has a bit set for each basic block transition that is exercised by t. We implemented a map $\{(\texttt{cksum}(i), f(i)) \mid t_i \in T\}$ that keeps track of the number of generated (and potentially discarded) inputs for each exercised path.

[7] Manual analysis and patching of 1.2k plus unique crashes took much time and hence was done for one program.

[8] http://www.openwall.com/lists/oss-security/2016/05/05/3

Measure of #crashes. AFL defines *unique crash* as follows. If two crashing inputs exercise a path in the same "bucket", then both inputs effectively expose the same unique crash.

Experimental Infrastructure. We ran our experiments on a 64-bit machine with 40 cores (2.6 GHz Intel® Xeon® E5-2600), 64GB of main memory, and Ubuntu 14.04 as host OS. We ran each experiment *at least eight times* for six or 24 hours. We ran 40 experiments simultaneously, that is, one experiment was run on one core. For each experiment, only one seed input is provided — the empty file. Time is measured using unix time stamps. We tested `nm -C`, `objdump -d`, `readelf -a`, and the others without options.

5.2 General Results

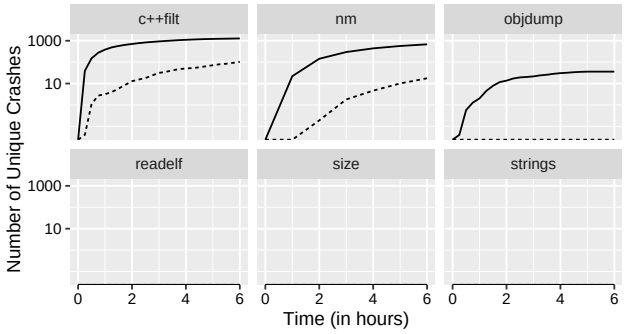

Figure 6: #Crashes over time (on a log-scale) for AFLFast (solid line) vs. AFL (dashed line)

Crashes over time. After 6h, AFLFast found one and two *orders of magnitude* more unique crashes than AFL in `c++filt` and `nm`, respectively.[9] AFLFast found 30 unique crashes in `objdump` where AFL found no crash at all. None of the fuzzers found a crashing input for the remaining three studied tools in any of eight runs of six hours. For each tool, the number of crashes found over time is shown in Figure 6. In what follows, we investigate the unique crashes generated for `nm` with a 24 hour budget in more details.

Vulnerabilities in `nm`. On average, AFLFast exposes the CVEs seven (7) times faster than AFL and exposes three (3) CVEs that are not exposed by AFL in any of eight runs in 24 hours. AFLFast exposes all vulnerabilities in 2h17m, on average while AFL would require more than 12h30m. The first three rows of Figure 7 show the results for the vulnerabilities in the `nm` tool in more details. Each facet compares AFLFast on the left hand-side and AFL on the right hand side using a box plot with a jitter overlay. In all of eight runs, AFLFast consistently and significantly outperforms classic AFL. The average time to first exposure is shown in Figure 8. All vulnerabilities are exposed within the first six hours. The exponential power schedule and improved search strategies clearly boost the efficiency of the state-of-the-art coverage-based greybox fuzzer.

Bugs in `nm`. AFLFast finds two buffer overflows seven (7) times faster than AFL. AFLFast also exposes a third bug which is not exposed by AFL at all. The three overflows are invalid reads and unlikely to be exploitable. The last row of Figure 1 shows more details. Again, our extension consistently outperforms the classic version of AFL.

[9]Notice the logarithmic scale in Figure 6.

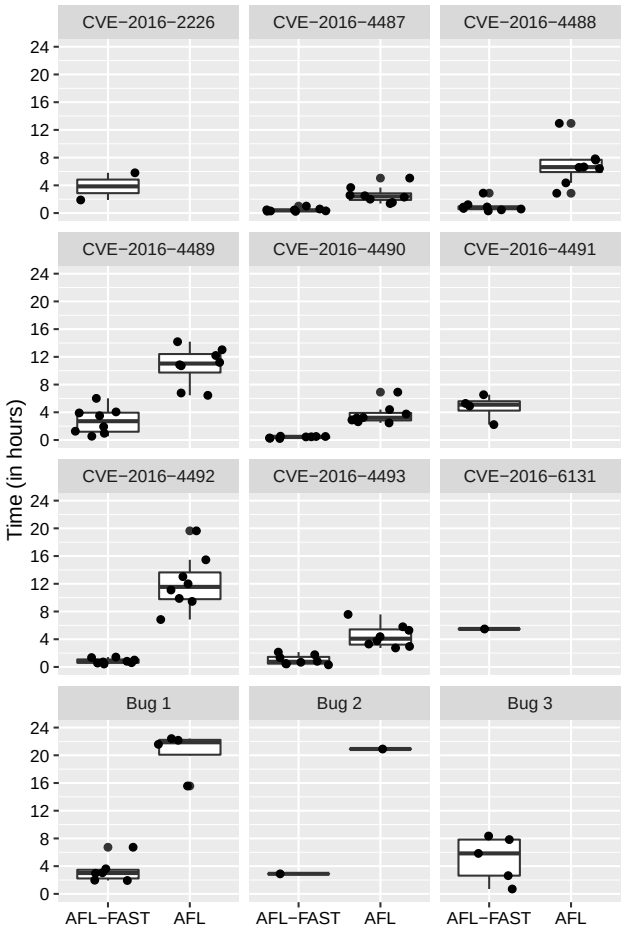

Figure 7: Time to expose the vulnerability.

Vulnerability	AFL	*AFL-Fast*	Factor
CVE-2016-2226	> 24.00 h	3.85 h	N/A
CVE-2016-4487	2.63 h	0.46 h	5.8
CVE-2016-4488	6.92 h	0.98 h	7.0
CVE-2016-4489	10.68 h	2.78 h	3.8
CVE-2016-4490	3.68 h	0.41 h	9.1
CVE-2016-4491	> 24.00 h	4.74 h	N/A
CVE-2016-4492	12.18 h	0.87 h	14.1
CVE-2016-4493	4.48 h	1.00 h	4.5
CVE-2016-6131	> 24.00 h	5.48 h	N/A
Bug 1	20.43 h	3.38 h	6.0
Bug 2	20.91 h	2.89 h	7.2
Bug 3	> 24.00 h	5.07 h	N/A

Figure 8: Time to expose the vulnerability.

Independent Evaluation. We note that our collaborators, Team Codejitsu at DARPA Cyber Grand Challenge (CGC), evaluated both AFL and AFLFast on all 150 benchmark programs that are provided as part of the CGC. On these binaries, AFLFast exposes errors 19x faster than AFL, on average. In one run, AFL exposed four errors that are not exposed by our extension. However, AFLFast exposed seven errors that are not exposed by AFL. Team Codejitsu integrated AFLFast in their bot Galatica to prove vulnerabilities in the other teams' binaries. Galactica went on to take 2$^{\text{nd}}$ place in the CGC finals *in terms of number of bugs found.* A thorough discussion and reflection of the CGC experience will not be covered in this article. However, we think that Codejitsu's success demonstrates the potential of AFLFast.

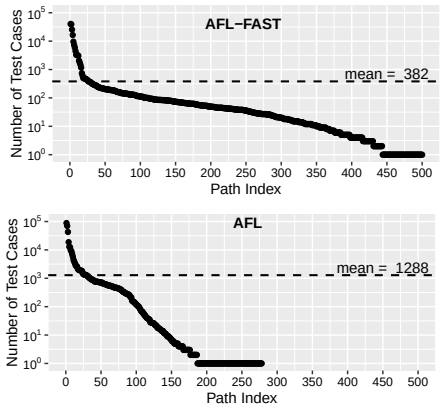

Figure 9: #Fuzz exercising a path (on a log-scale) after running AFL for 10 minutes on the `nm`-tool.

Low-frequency Paths. In this paper, we argue that the fuzzing time is better spent exploring low-frequency paths. Firstly, we believe that low-frequency paths are more likely to be exercised by valid inputs that stress different behaviors of the program. Secondly, less time is wasted fuzzing high-frequency paths that are exercised by most fuzz anyways. Finally, it allows the coverage-based greybox fuzzer to efficiently discover more paths per generated input. As we can see in Figure 9, indeed our heuristics generate more fuzz for low-frequency paths and less fuzz for high-frequency paths. In 10 minutes, AFLFast discovered twice as many paths as AFL. For AFLFast only 10% of the discovered (low-frequency) paths are exercised by just one input while for AFL, 30% are exercised by just one input. The mean amount of generated test inputs per path is about three times higher for AFLFast. This clearly demonstrates the effectiveness of our heuristics in exploring a maximal number of (low-frequency) paths while expending minimum energy.

5.3 Comparison of Power Schedules

Earlier, we introduced two constant and four monotonous power schedules. AFL adopts a constant power schedule and assigns a fairly high amount of energy. Basically, the same input will get the same performance score the next time it is fuzzed. This is the exploitation-based constant schedule (exploit). To understand the impact of our choice to start with a reduced fuzzing time per input, we also investigate an exploration-based constant schedule (explore) that assigns a fairly low and constant amount of energy. The monotonous schedules increase the fuzzing time in a linear, quadratic, or exponential manner. Specifically, AFLFast implements an exponential schedule.

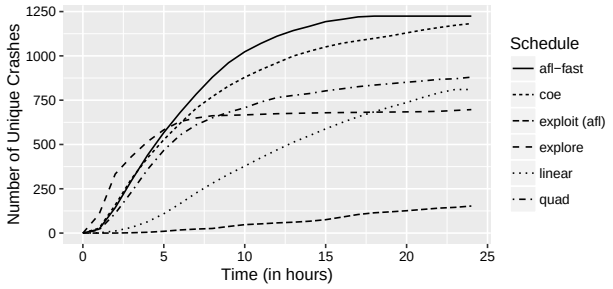

Figure 10: #Crashes over Time (Schedules).

Results. The exponential schedule that is implemented in AFLFast outperforms all other schedules. The cut-off exponential schedule (coe) performs only slightly worse than AFLFast. After 24 hours, both schedules (fast and coe) exposed 50% more unique crashes than the other three (linear, quad, and explore). Interestingly, the exploration-based constant schedule (explore) starts off by discovering a larger number of crashes than any of the other schedules; it fuzzes each input quickly and swiftly moves on to the next. However, this strategy does not pay off in the longer run. After 24 hours, it performs worse than any of the other schedules (except AFL's exploitation-based constant schedule). The quadratic schedule (quad) starts off revealing a similar number of unique crashes as AFLFast but at the end of the 24 hour budget it performs comparably to the other two (linear and explore).

5.4 Comparison of Search Strategies

Our search strategies prioritize inputs that have not been fuzzed very often (small $s(i)$) and inputs that exercise low-frequency paths (small $f(i)$). In the following, we investigate two strategies targeting the implementation of PERF_SCORE and CHOOSENEXT in Algorithm 1. *Strategy 1* designates as favourites $t_i \in T$ where $s(i)$ and $f(i)$ are small, and then where execution time, transition coverage, and creation time are minimal.[10] Without Strategy 1, AFLFast (like AFL) designates as favorites $t_i \in T$ where execution time, transition coverage, and creation time are minimal. *Strategy 2* chooses the next input t_i from the queue where $s(i)$ and $f(i)$ are minimal and t_i is a favourite. Without Strategy 2 AFLFast (like AFL) chooses the next input from the queue that is marked as favourite. All strategies are run with the exponential power schedule.

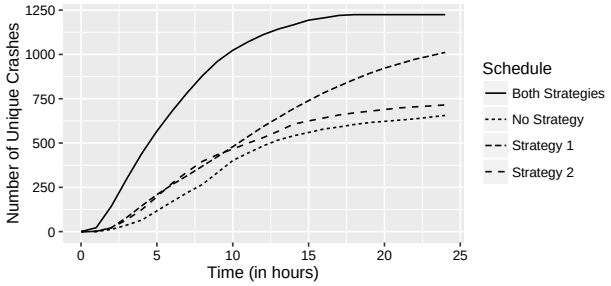

Figure 11: #Crashes over Time (Search Strategies).

Results. The combination of both strategies is significantly more effective than any of the strategies individually. Until about 12 hours the other strategies perform very similarly. After 24 hours as individual strategy, strategy 1 which changes how AFL designates the favourite is more effective than strategy 2 and no strategy in the long run. As individual strategy, the strategy 2 which changes the order in which test inputs are chosen from the queue seems to be not effective at all. It performs similarly compared to running AFLFast without any strategies (comparable to AFL but with exponential power schedule). However, after 24 hours, AFLFast with both strategies exposes almost twice as many unique crashes as AFLFast with no strategy or with only strategy 1.

[10]For more details see Section 4.3.

5.5 Result Summary

We evaluated AFLFast and several schedules plus search strategies on the GNU binutils. The exponential schedule outperforms all other schedules while our search strategies turn out to be effective. In eight runs of six hours, AFLFast with an exponential schedule found an average of more than one order of magnitude more unique crashes than AFL for the tools `nm` and `c++filt`; it found crashing inputs for `objdump` where AFL did not expose any crashes at all. In eight runs of 24 hours, AFLFast found 6 vulnerabilities in `nm` 7x faster than AFL and exposed 3 vulnerabilities that were not exposed by AFL. AFLFast also exposes two bugs in `nm` (that are unlikely exploitable) about seven times faster than AFL and exposed one bug that is not exposed by AFL. An independent evaluation of Team Codejitsu on all 150 binaries that are provided in the benchmark for the Cyber Grand Challenge establishes similar results. On average, AFLFast exposes an error 19 times faster than AFL and also exposes 7 errors that are not found by AFL, at all.

6. RELATED WORK

Several techniques [30, 17, 6, 22] have been proposed to increase the efficiency of automated fuzzing. An important optimization pertains to selecting the seed inputs wisely from a wealth of inputs [17]. Our work makes no assumptions about the existance seed inputs; we seeded our experiments with the empty file. However, Coverage-based Greybox Fuzzing (CGF) would clearly benefit from a smart seed selection if many seed files are available. Others suggest to use program analysis to detect dependencies among the bit positions of an input [6]. For instance, the image width occupies four bytes in the PNG image file format which are best modified together. The dependency analysis allows to fuzz such dependent bytes as a group. In our work, we do not change the mutation operators or ratio. Woo et al. [30] recognize the exploration-exploitation trade-off between fuzzing an input for a shorter versus a longer amount of time. They proceed to model blackbox fuzzing as a multi-armed bandit problem where the seed's "energy" is computed based on whether or not it has exposed a (unique) crash in any previous fuzzing iteration. So, the fuzzer is effectively biased towards generating more crashing inputs for already known errors. In our work, there is no such bias. Instead, we direct the search towards low-frequency paths in order to stress more of the program's behavior in the same time.

Symbolic execution-based whitebox fuzzers can generate files that stress low-frequency paths. Probabilistic symbolic execution [10] uses model counting to compute the probability that a random input exercises a given path. Symbolic execution is very effective because it enumerates paths essentially independent of their "frequency" and because it can be directed towards "dangerous" program locations [5, 8, 11, 2]. It can generate the specific values that are needed in order to negate an if-condition and exercise the alternative branch. Taint-based fuzzing [9, 29] is a directed whitebox fuzzing technique. It exploits classical taint analysis to localize parts of the input which should be marked symbolic. For instance, it marks portions of the input file as symbolic that control arguments of executed and critical system calls. Model-based Whitebox Fuzzing [16] leverages an input model to synthesize and "transplant" complete data chunks to exercise so called *critical* branches that are only exercised if a certain data chunk is present in the input file. However, symbolic execution-based techniques rely on program analysis and constraint solving which hampers their scalability. Imprecisions during lifting of the program binary and during the encoding of the path constraints hamper their applicability. In contrast, CGF completely relinquishes program analysis for the sake of scalability with tremendous success in the vulnerability detection practice [27].

Colleagues have combined lightweight blackbox/greybox fuzzers and symbolic execution-based whitebox fuzzers to get the best of both worlds [19, 15]. For instance, HybridFuzz first runs symbolic execution to generate inputs leading to "frontier nodes" and then passes these inputs to a blackbox fuzzer. In contrast, Driller [19] begins with AFL and seeks help from symbolic execution when it "gets stuck", for instance, to generate a magic number. Our monotonous power schedules allow to employ expensive symbolic execution for seeds/states with a sufficiently high energy.

Markov chains can model a variety of random processes in fuzz testing. Markov Chain Monte Carlo Random Testing (MCMC-RT) uses a Markov Chain Monte Carlo (MCMC) method to leverage knowledge about an input's probability to reveal an error. However, MCMC-RT is not entirely scalable because it maintains this probability for *every* input in the program's input space. While CGF can be well explained as Markov chain, it does not actually maintain the chain or any probabilities in-memory. While MCMC-RT is biased towards revealing suspected or known errors, CGF can expose unknown errors that hide deep in the program. The bias of boosted CGF is towards low-frequency paths. Chen et al. [7] utilize MCMC to leverage knowledge about a mutation operator's effectiveness. Operators that have been shown to be more effective in previous fuzzing iterations are chosen with greater probability during fuzzing. Sparks et al. [18] model program control-flow as Markov chain to prioritize seeds that exercise less explored paths. In contrast, we use Markov chains to *explain* why it is more efficient to smartly control the time spent fuzzing a seed and which seed to fuzz next *without* program analysis.

7. CONCLUSION

While symbolic execution-based techniques have gained prominence, their scalability has not approached those of blackbox or greybox fuzzers. While blackbox and greybox techniques have shown effectiveness, the limited semantic oversight of these techniques do not allow us to explain the working of these techniques even when they are effective.

In this work, we take a state-of-the-art greybox fuzzer AFL which keeps track of path identifiers. We enhance the effectiveness and efficiency of AFL in producing crashes, as evidenced by our experiments and those of our collaborators. AFLFast, our extension of AFL exposes an order of magnitude more unique crashes than AFL in the same time budget. Moreover, AFLFast can expose several bugs and vulnerabilities that AFL cannot find. Other vulnerabilities AFLFast exposes substantially earlier than AFL.

More importantly, we provide an explanation of the enhanced effectiveness by visualizing CGF as the exploration of the state space of a Markov chain. We observe that existing CGF tools much too often visit states in high-density regions. We have devised and investigated several strategies to force the CGF tool to visit more states that are otherwise hidden in a low-density region and to generate less inputs for states in a high-density region.

8. ACKNOWLEDGMENTS

This research was partially supported by a grant from the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

9. REFERENCES

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, 2005.

[2] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 334–344, 2013.

[3] M. Böhme and S. Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, April 2016.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[6] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 725–741, 2015.

[7] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of jvm implementations. In *PLDI' 16*, pages 85–99, 2016.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS XVI*, pages 265–278, 2011.

[9] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, 2009.

[10] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 166–176, 2012.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[12] S. Kirkpatrick, C. Jr. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[13] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[14] J. R. Norris. *Markov Chains (Cambridge Series in Statistical and Probabilistic Mathematics)*. Cambridge University Press, July 1998.

[15] B. S. Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. In *Master's thesis, School of Computer Science, Carnegie Mellon University*, 2012.

[16] V.-T. Pham, M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 552–562, 2016.

[17] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 861–875, 2014.

[18] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *23d Annual Computer Security Applications Conference (ACSAC)*, pages 477–486, 2007.

[19] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS '16*, pages 1–16, 2016.

[20] Tool. Afl binary instrumentation. https://github.com/vrtadmin/moflow/tree/master/afl-dyninst. Accessed: 2016-05-13.

[21] Tool. Afl vulnerability trophy case. http://lcamtuf.coredump.cx/afl/#bugs. Accessed: 2016-05-13.

[22] Tool. American fuzzy lop (afl) fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2016-05-13.

[23] Tool. Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/. Accessed: 2016-05-13.

[24] Tool. Pulling jpegs out of thin air. https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html. Accessed: 2016-05-13.

[25] Tool. SPIKE Fuzzer Platform. http://www.immunitysec.com. Accessed: 2016-05-13.

[26] Tool. Suley Fuzzer. https://github.com/OpenRCE/sulley. Accessed: 2016-05-13.

[27] Tool. Symbolic execution in vulnerability research. https://lcamtuf.blogspot.sg/2015/02/symbolic-execution-in-vuln-research.html. Accessed: 2016-05-13.

[28] Tool. Zzuf: multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf. Accessed: 2016-05-13.

[29] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 497–512, 2010.

[30] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 511–522, 2013.