Coverage-based Greybox Fuzzing as Markov Chain

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury

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 several schedules 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. We compared AFLFast to the symbolic executor Klee. In terms of vulnerability detection, AFLFast is significantly more effective than Klee on the same subject programs that were discussed in the original Klee paper. In terms of code coverage, AFLFast only slightly outperforms Klee while a combination of both tools achieves best results by mitigating the individual weaknesses.

Index Terms—vulnerability detection, fuzzing, path exploration, symbolic execution, automated testing

1 INTRODUCTION

RECENTLY, there has been a controversial debate about the efficiency of symbolic execution-based whitebox fuzzers versus more lightweight greybox fuzzers [2], [3]. Symbolic execution is a systematic effort to stress different behaviors and thus considerably more effective. Yet, today most vulnerabilities were exposed by particularly lightweight fuzzers that do not leverage any program analysis.

It turns out that even the most effective technique is less efficient than blackbox fuzzing if the time spent generating a test case takes too long [4]. 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 for one test depends on the program’s execution time rather than its size or complexity. 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 [5], has been shown to generate valid image files (JPEGs) from an initial seed that is virtually empty [6], and has also been integrated with symbolic execution (which helps where AFL “gets stuck”) [3]. Clearly, increasing the efficiency of fuzzers, like AFL, has a real and practical impact on vulnerability detection.

“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 [2]

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 $\pi$ 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 $\pi$. So, independent of the initial seed image files, an above-average amount of
fuzz is bound to exercise that path \( \pi \) which rejects invalid inputs. Informally, we call \( \pi \) a high-frequency path.

In this paper, we propose several strategies to systematically bias the fuzzer towards exercising 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. 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 a 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 \( p_{ij}^0 \). Note, however, that in practice \( p_{ij} \) is clearly unknown. The energy of a state is controlled by a predefined 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. Every time the seed is chosen, about 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 initial 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 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. To estimate the fuzzer’s probability of exercising a path \( i \), AFLFAST uses the maximum-likelihood estimator \( \hat{p}_i = \frac{f(i)}{n} \) where \( f(i) \) is the number of fuzz exercising \( i \) and \( n \) is the total number of fuzz generated throughout the fuzzing campaign.

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 on average). Since we do not modify the mutation operator 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 effectively exposes exactly the same vulnerabilities as AFL – only significantly faster.

In summary, we argue that the effectiveness 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.

To understand the difference in practice, we compared KLEE and AFLFAST in terms of vulnerability detection and coverage achieved using the same subjects and settings that were used in the original KLEE-paper. KLEE exposes six of the nine KLEE-benchmark errors quicker than AFLFAST. However, AFLFAST discovered five new bugs that KLEE did not expose. This indicates that KLEE is able to detect certain types of errors quickly while it is unable to detect some other types of errors in the 1-hour time frame. Investigating the reason, we found that KLEE requires a constraint-based vulnerability detection mechanism and presumes the completeness of the constraint-encoding as well as the completeness of the environment modelling. In contrast, AFLFAST merely executes the program and reports any crashes, e.g., via runtime checking. In terms of code coverage, we found that AFLFAST performs slightly better than KLEE, covering an average of 82% of executable lines of code in 50 programs versus 78% for KLEE. However, we believe that a combination of both techniques is even more powerful, using their individual strengths to mitigate each others weaknesses.

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

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.

- **Comparison to Symbolic Execution.** We empirically compare the popular symbolic execution engine KLEE with our implementation of coverage-based greybox fuzzing, both in terms of vulnerability detection as well as in code coverage achieved.

- **Tool.** We published AFLFAST as a fork of AFL. Since the first release, AFLFAST has had considerable impact in software security practice, being featured in several blog posts and hundreds of tweets. AFLFAST was used by Team Codejitsu who came in 2nd in terms of number of bugs found at the DARPA Cyber Grand Challenge: https://github.com/mboehme/aflfast

The remainder of this article is structured as follows. After the introduction in Section 1, we introduce some background in Section 2. The Markov chain model is introduced in Section 3 and used to discuss opportunities to boost the efficiency of CGF in Section 4. We compare our implementation of boosted CGF (i.e., AFLFAST) with vanilla CGF (i.e., AFL) in Section 5 and with a symbolic execution engine (KLEE) in Section 6. After a discussion of related work in Section 8, we conclude in Section 9.

## 2 BACKGROUND

### 2.1 Coverage-based Greybox Fuzzing

**Fuzz —** an automated random testing tool was first developed by Miller et al. [8] in early 1990s to understand the reliability of UNIX tools. Since then, fuzzing has evolved substantially, becomes widely adopted into practice, and exposed serious vulnerabilities in many important software programs [7], [10], [11], [12]. There are three major categories depending on the degree of leverage of internal program structure: black-box fuzzing only requires the program to execute [9], [10], [13] while white-box fuzzing [7], [14], [15], [16] requires binary liting 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)** [12] 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 Figure 1.

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

**Fig. 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 over shared_mem[] is used as the path identifier.

**Algorithm 1 Coverage-based Greybox Fuzzing**

**Input:** Seed Inputs \(S\)

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

**Output:** Crashing Inputs \(T_k\)

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 and Figure 2 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.

2.2 Markov Chain
A Markov chain is a stochastic process that transitions from one state to another \[^{19}\]. 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 $P(X_0 = i)$.

The probability matrix $P = (p_{ij})$ specifies the transition rules. If $|S| = N$, then $P$ 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 $t$,

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

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 $\pi$ 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 chains approach the stationary distribution within reasonable time—indepenent 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 \[^{20}\] 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-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 the fuzzer fuzzing any path \( t \in T \). 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 \( t_i \), 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 any \( t \in T \) (i.e., \( i \in S^- \)), then \( p_{ii} = 1 - \sum_{j \in T} p_{ij} \) and \( p_{ij} = p_{ji} \) for all \( t_j \in T \). In other words, 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 \( t_i \) an input that exercises (undiscovered) path \( i \). We also assume that \( i \in S^- \) has no other undiscovered neighbors.

4. An input \( t_i \) is randomly mutated using mutate input on \( t_i \) in Algorithm 1.
are valid and interesting. Basically, almost each valid input
would exercise different behavior. Hence, in this paper we
devises 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 pro-
gram properties, such as the worst-case or average exe-
cution time or energy consumption. There exist several
Markov Chain Monte Carlo (MCMC) methods, like Simu-
lated Annealing that offer guarantees on the conver-
gence to the actual value. In the context of vulnerability
research, the Markov chain model allows to explain the chal-
enges 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 Figure 4 which takes as input a 4-character word
and crashes for the input “bad!”.

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

Fig. 4. 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 4 to modify a seed input 
\( s = (c_0, c_1, c_2, c_3) \) 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 256 ASCII
characters. For example, the word “bill” exercises Path 1.
With probability 1/4, MI chooses the second character \( c_1 \)
and with probability 1/256 it chooses the letter ‘a’ for the
substitution. With a total probability of 210, MI generates
the word “ball” from “bill” as the next test input which
exercises Path 2.

![Figure 5. Markov chain for motivating example](image-url)

Figure 5 represents the simplified transition matrix \( p_{ij} \)
as a state diagram. 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 210 = 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 (1/4 – 2−10) 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!.

5. 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 \( \text{choosenext} \) and generates as many inputs as determined by \( p = \text{assignenergy}(t) \) in Alg. 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 \( i \) be a path such that \( i \in S^+ \) and let \( t_i \) be the seed input that exercises \( i \). Let \( X_{ij} \) be a random variable describing the number fuzz generated from \( t_i \) before discovering the path \( j \in S^- \). We solve for the expected value of \( X_{ij} \) using a recursive argument: The probability of discovering \( j \) with the first fuzz that is generated from \( t_i \) is \( p_{ij} \). The probability of not discovering \( j \) and requiring at least one moreuzz is \((1 - p_{ij})\). Formally,

\[
E[X_{ij}] = p_{ij} + (E[X_{ij}] + 1)(1 - p_{ij})
\]

In other words, in expectation the minimum energy that should be assigned to state \( i \in S^+ \) so that the fuzzer discovers the new state \( j \in S^- \) is \( 1/p_{ij} \). 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 6 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 \( E[X_{i01}] + E[X_{i12}] + E[X_{i23}] + E[X_{i34}] = 4 \cdot 2^{10} = 4k \) inputs to exercise the same vulnerability.

To make this more concrete, we experimentally determined the energy a single seed assigns to each path. 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 7 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., \( 2^9 \) inputs) exercises paths **** and b***, respectively (see Fig. 6 and Fig. 7 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 \( \pi_j \) 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 chosen from the queue \( T \) and b) the number of generated inputs \( f(i) \) that exercise \( i \). In fact, \( f(i)/n \) is the maximum-likelihood estimator of the CGF’s probability to generate an input that exercises \( i \) and thus approximates 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

\[
p(i) = \alpha(i) \quad \text{e.g., for AFL} \tag{1}
\]

where \( \alpha(i) \) is the CGF’s present implementation 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 6 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 \( \alpha(i)/\beta \) maintains 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

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

where \( \alpha(i) \) maintains 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 \( \mu \) is the mean number of fuzz exercising a discovered path

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

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.

<table>
<thead>
<tr>
<th>#Tests</th>
<th>State</th>
<th>Explored States</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>****</td>
<td>****</td>
</tr>
<tr>
<td>2</td>
<td>b***</td>
<td>***** , b***</td>
</tr>
<tr>
<td>3</td>
<td>b***</td>
<td>b*** , b*** , ba**</td>
</tr>
<tr>
<td>4</td>
<td>b***</td>
<td>b*** , b*** , ba**, bad*</td>
</tr>
</tbody>
</table>

Fig. 8. The crash is found after \( 2^{12} = 4k \) inputs were generated by fuzzing with a power schedule.

**Example.** Figure 8 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. 5). Now, the newly discovered state is assigned low energy \( \alpha(1) = 2^0 \). However, \( f(0) > 0 \) so that solely \( t_1 \) will be fuzzed in a similar fashion as \( t_0 \) until \( s(1) = 9, \alpha(1) = 2^9, 2^{10} \) test inputs have been generated by fuzzing \( t_1 \). Again, one test input is expected to start with “ba” and the state b** is discovered. Figure 8 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 6 would require one order of magnitude more test inputs on average \((256 k)\).

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

\[
p(i) = \min \left( \frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)} - 1}{f(i)}, M \right)
\]

(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 \( i \).

\[
p(i) = \min \left( \frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M \right)
\]

(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

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

(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 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 chosen 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.

5 Evaluation: Vanilla versus Boosted CGF

We compare AFLFast our implementation of the presented monotonous power schedules with AFL 1.94b the baseline implementation featuring a constant schedule.

5.1 Implementation: AFLFast 1.94b

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.

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 smallest 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 \( f(i) \).

While AFLFast has since been improved, the version we used for the experiments is still available at https://github.com/mboehme/aflfast/tree/AFLFast.old.

5.2 Vulnerabilities

We chose GNU binutils v2.26 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. 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).
All vulnerabilities were previously unreported and rated as medium security risk. We informed the maintainers, submitted patches, and informed the security community via the osssecurity mailing list. Mitre assigned nine (9) CVEs. At the time of writing, all patches have been accepted. 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 when trying to reverse-engineer the binary.

### Table 1: CVE-IDs and Exploitation Type

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-2226</td>
<td>Exploitable Buffer Overflow</td>
</tr>
<tr>
<td>CVE-2016-4487</td>
<td>Invalid Write due to a Use-After-Free</td>
</tr>
<tr>
<td>CVE-2016-4488</td>
<td>Invalid Write due to a Use-After-Free</td>
</tr>
<tr>
<td>CVE-2016-4489</td>
<td>Invalid Write due to Integer Overflow</td>
</tr>
<tr>
<td>CVE-2016-4490</td>
<td>Write Access Violation</td>
</tr>
<tr>
<td>CVE-2016-4491</td>
<td>Various Stack Corruptions</td>
</tr>
<tr>
<td>CVE-2016-4492</td>
<td>Write Access Violation</td>
</tr>
<tr>
<td>CVE-2016-4493</td>
<td>Write Access Violation</td>
</tr>
<tr>
<td>CVE-2016-6131</td>
<td>Stack Corruption</td>
</tr>
</tbody>
</table>

### Table 2: Subjects from Binutils

<table>
<thead>
<tr>
<th>Utility</th>
<th>Parameters</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>c++filt</td>
<td>@</td>
<td>27kB</td>
</tr>
<tr>
<td>nm</td>
<td>-C @</td>
<td>41kB</td>
</tr>
<tr>
<td>objdump</td>
<td>-d @</td>
<td>36kB</td>
</tr>
<tr>
<td>readelf</td>
<td>-a @</td>
<td>49kB</td>
</tr>
<tr>
<td>size</td>
<td>@</td>
<td>32kB</td>
</tr>
<tr>
<td>strings</td>
<td>@</td>
<td>28kB</td>
</tr>
</tbody>
</table>

**Subjects.** The specific utilities and command line parameters are shown in Table 2. The parameter @ is a placeholder for the file that the fuzzer generates as input.

**Measure of #paths.** AFL maintains a unique path identifier 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 \{ (cksum(i), f(i)) | t_i \in T \} that keeps track of the number of generated (and potentially discarded) inputs for each exercised path.

**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.

### 5.3 General Results

**Crashes over time.** After 6h, AFLFAST found one and two orders of magnitude more unique crashes than AFL in `c++filt` and `nm`, respectively. 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 9. In what follows, we investigate the unique crashes generated for `nm` with a 24 hour budget in more details.

**Table 3: Time to expose the vulnerability.**

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>AFL</th>
<th>AFL-Fast</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-2226</td>
<td>24.00 h</td>
<td>3.85 h</td>
<td>N/A</td>
</tr>
<tr>
<td>CVE-2016-4487</td>
<td>2.63 h</td>
<td>0.46 h</td>
<td>5.8</td>
</tr>
<tr>
<td>CVE-2016-4488</td>
<td>6.92 h</td>
<td>0.98 h</td>
<td>7.0</td>
</tr>
<tr>
<td>CVE-2016-4489</td>
<td>10.68 h</td>
<td>2.78 h</td>
<td>3.8</td>
</tr>
<tr>
<td>CVE-2016-4490</td>
<td>3.68 h</td>
<td>0.41 h</td>
<td>9.1</td>
</tr>
<tr>
<td>CVE-2016-4491</td>
<td>24.00 h</td>
<td>4.74 h</td>
<td>N/A</td>
</tr>
<tr>
<td>CVE-2016-4492</td>
<td>12.18 h</td>
<td>0.87 h</td>
<td>14.1</td>
</tr>
<tr>
<td>CVE-2016-4493</td>
<td>4.48 h</td>
<td>1.00 h</td>
<td>4.5</td>
</tr>
<tr>
<td>CVE-2016-6131</td>
<td>24.00 h</td>
<td>5.48 h</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**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 found vulnerabilities in 2h17m, on average while AFL would require more than 12h30m. The first three rows of Figure 10 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 Table 3. 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.

---

6. [http://www.openwall.com/lists/oss-security/2016/05/05/3](http://www.openwall.com/lists/oss-security/2016/05/05/3)
7. Notice the logarithmic scale in Figure 9
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 Table 1 shows more details. Again, our extension consistently outperforms the classic version of AFL.

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 Galactica to prove vulnerabilities in the other teams’ binaries. Galactica went on to take 2nd 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.

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 11, 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.

5.4 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.
5.5 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 assignEnergy 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. 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.

5.6 Result Summary

We evaluated AFLFAST and several schedules plus search strategies on the GNU binutils v2.26. 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 that are 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 Symbolic-Execution-based Whitebox Fuzzing vs Coverage-based Greybox Fuzzing

We compare AFLFAST our boosted coverage-based greybox fuzzer implementing the exponential schedule with the most recent version of KLEE v1.2-094a2183[7] a widely-used state-of-the-art symbolic-execution-based whitebox fuzzer—both in terms of vulnerability detection and in terms of code coverage. For the comparison in terms of vulnerability detection, we use the same subjects as the authors of KLEE when it was originally introduced, GNU Coreutils v6.10-1440ca24. Having exposed previously unreported errors that still existed in the most recent version, we were motivated to repeat this experiment on the most recent version, Coreutils v8.25-f1799198. For the comparison in terms of code coverage, we use the most recent version of Coreutils v8.25-f1799198.

6.1 Conceptual Comparison

Symbolic execution is designed for efficient path exploration, the systematic enumeration of the paths in a program. For each path in the control-flow graph of a program, the symbolic execution engine can construct a path constraint, a quantifier-free first-order logic formula that is satisfied by all inputs exercising this path. The path constraint is constructed as the conjunction of branch constraints. One branch constraint is added to the path constraint for each conditional instruction along the control-flow-path, deciding whether the true or false branch should be exercised next. The resulting path constraint is passed to a SMT solver to provide an input that satisfies the path constraint and thus exercises the path. KLEE[7] is a symbolic-execution engine that implements several search heuristics to explore paths particularly that yield a high degree of code coverage quickly.

In contrast, blackbox fuzzers are not at all designed for path exploration. They treat the program as a blackbox whence there is no notion of program path. However, blackbox fuzzers can generate an enormous amount of inputs in
a very short time. In earlier work, we showed that even the
most effective fuzzers is less efficient than blackbox fuzzing
if generating a test input takes relatively too long [3]. Since
the blackbox fuzzer generates so many inputs (compared to
a symbolic execution engine), many paths will be explored
quickly in a collateral, incidental manner. However, the
blackbox fuzzer’s ability to discover new paths quickly
decelerates while the symbolic execution engine keeps on
discovering new paths steadily. So, the symbolic execution
engine outperforms the blackbox fuzzer when more time is
available.

Now, a coverage-based greybox fuzzer (CGF) is much
better at path exploration than a blackbox fuzzer. The
objective of a CGF is explicitly to discover as many new and
interesting paths as possible. A CGF leverages lightweight
instrumentation to identify which path is exercised by an
input. It uses the inputs that were retained because they
exercised new and interesting paths to explore their neigh-
borhood and discover further new and interesting paths
that were otherwise difficult to reach from the original
seed inputs. While symbolic execution uses program analy-
sis, tainting, and constraint solving, a CGF still uses only
random mutations of the seed inputs, retaining much of
the efficiency of blackbox fuzzers. Our power schedules
improve on the capability to explore a maximal number of
paths per unit time.

Hence, we compare KLEE and AFLFAST head to head
to determine which one is better at path exploration, code
coverage, and exposing errors. We leverage the same bench-
marks as the original KLEE paper. However, we note that
such tool comparisons should always be taken with a grain
of salt. An empirical evaluation is always comparing only
the implementations of two concepts rather than the con-
cepts themselves. Improving the efficiency or extending the
search space may only be engineering effort unrelated to
the concept [22]. We make a conscious effort to explain
the observed phenomena and distinguish conceptual from
technical origins.

### 6.2 Fuzzing Command Line Utilities using AFLARGV

The GNU Coreutils [23] is a collection of more than 50
widely used command-line tools, including cp to copy files
and mkdir to create new directories. An overview of the
utilities for a more recent version is shown in Table 4.

<table>
<thead>
<tr>
<th>Min</th>
<th>Max</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFF</td>
<td>0x02</td>
<td>0x00</td>
</tr>
<tr>
<td>0x66</td>
<td>0x00</td>
<td>0x09</td>
</tr>
</tbody>
</table>

A file is a sequence of bytes split into chunks.

Most utilities have mandatory and optional command line
parameters and process one or more files. Given lower and
upper bounds on the number and size of options and files,
KLEE is able to synthesis both options and files during the
symbolic execution. However, AFLFAST (just like AFL)
can only generate a single file as program input.

To facilitate the proper testing of the Coreutils command-
line utilities, we implemented AFLARGV a simple tool that
takes (i) lower and upper bounds on the number and size
of command-line options and files and (ii) an input file,
and disassambles the given file into command-line op-
tions and the required number of files. AFLARGV allows
to specify the upper and lower bounds using the same
format as KLEE. The AFLARGV arguments -sym-args
<MIN> <MAX> <N> would specify between MIN and MAX
number of command-line options of maximum length N
characters. The AFLARGV arguments -sym-files <MIN>
<MAX> <N> would specify between MIN and MAX number
of files of maximum N bytes. There can be one argument
-message that specifies a file of arbitrary length. For instance,
to use AFLFAST with AFLARGV to fuzz the utility md5sum
with up to one argument of maximum length four and a file
of arbitrary length, we would execute:

```bash
$ afl-fuzz -i in -o out \n AFLargv -sym-args 0 1 4 -file md5sum
```

The file format leverages fixed-length data chunks to accom-
modate the actual values and size for the specified options
and files. For instance, AFLFAST generates the following
fuzz as input to AFLARGV:

```
0xFF 0x02 0x2D 0x66
```

Via the AFLARGV arguments, we specified up to one
command-line option of maximum four characters. The first
two bytes (1) specify that this optional parameter is indeed
used and that the parameter is two characters long. The
following four bytes (2) specify the ASCII code of the actual
argument. Only two bytes are interpreted (0x2D 0x66
is ASCII code for "-f") . The remaining two bytes are ignored.
Via the AFLARGV arguments, we specified a file of
arbitrary length. Hence, everything beyond the sixth byte
(3) is interpreted as file content.

AFLFAST is oblivious to AFLARGV. As far as the fuzzer
is concerned, it generates inputs with the objective to dis-
cover a maximum number of paths per unit time. Given the
generated fuzz, AFLARGV synthesizes the files and passes
the corresponding command-line parameters to the tool
under test which (as per normal) provides feedback on the
discovered paths. In our example, AFLARGV generates a file
t1.txt and calls md5sum -f t1.txt.

### 6.3 Experimental Setup

We used the same subjects and experimental parameters
as provided in the original KLEE paper [7] in order to
mitigate experimenter bias and allow for a fair comparison.
We run each fuzzing campaign for one (1) hour using the
same experimental infrastructure that was discussed earlier
(Sec. 5.2). We start KLEE and AFLFAST with the following
command-line options (as provided by the authors):

```
-sym-args 0 1 10 -sym-args 0 2 2 -sym-files 1 8
```

We start AFLFAST with no options and an empty file.
Technically, we assembled a single, “empty” seed file rep-
resenting the minimum number of options, all of length
zero characters, and the minimum number of files, all of
size zero bytes. AFLARGV also allows to assemble such a
file from a provided set of arguments and files according to
the required file format. This would be important when we
needed to seed the fuzzer using test cases from an existing
regression test suite. However, we do not actually utilize
that functionality in our experiments.

We measure effectiveness by executing the generated
inputs on the utilities compiled with AddressSanitizer [24].


10. Tutorial at https://klee.github.io/tutorials/testing-coreutils
6.4 Vulnerability Detection

In this experiment, we compare the symbolic execution-based whitebox fuzzer KLEE and our boosted greybox fuzzer AFLFAST in terms of vulnerability detection. We use the KLEE benchmark, a collection of bugs that were reported in the original KLEE paper [7].

First, we measure the time it takes to expose these errors (efficiency). Second, we discuss bugs found only by AFLFAST but not by KLEE (effectiveness). Finally, encouraged by our findings, we conducted an experiment using the most recent version of Coreutils where we compare AFLFAST and KLEE in terms of previously unreported bugs found (effectiveness).

KLEE AFLFAST How to Reproduce
09:48 > 1h md5sum -c t1.txt
16:49 00:20 mkdir -Z a b
09:30 00:20 mkfifo -Z a b
00:02 00:24 mknnod -Z a b p
00:16 01:53 paste -d\ \ abcdefghi\klmnopqrstuvwxyz
16:17 21:56 pr -e t2.txt
00:04 07:14 ptx -F\ \ abcdefghi\klmnopqrstuvwxyz
00:03 01:31 seq -f %0 i
05:00 tac > -t t3.txt

where files t1.txt: "\t\nKFILE (*

Fig. 14. Time to expose the KLEE benchmark bugs. AFLFAST is fairly deterministic. So, we report the results for one run (as min:sec). AFLFAST is a random approach. Hence, we report the mean over ten runs.

Results (Efficiency). On the benchmark, AFLFAST is more efficient than AFLFAST for six out of nine bugs (Figure 14). However, on average AFLFAST requires much less time to expose an error than KLEE. Specifically, the mean time-to-exposure is 4:49 for AFLFAST and 6:51 for KLEE. Moreover, we note that the AFLFAST benchmark consists of bugs that were explicitly discovered and reported by the authors of KLEE, which introduces a selection bias: AFLFAST might be best in discovering those errors that it did discover (but not in discovering those that it did not discover).

KLEE outperforms AFLFAST in terms of efficiency on the KLEE benchmark bugs.

How to reproduce Crash Location Symptom
pr -155555555 -5 prc1918 Memory exhausted
echo "\n" | ptx -W.+ -O regexexec.c:2354 Buf. Overflow (W)
echo "\n" | ptx -Fw regex_internal.c:3940 Buf. Overflow (R)
echo "\n" | ptx -W++ -A regexexec.c:3780 Buf. Overflow (R)
echo "\n" | ptx -W\ -A regexexec.c:3231 Buf. Overflow (R)

where file A: "\n"

Fig. 15. Bugs found only by AFLFAST in the same subjects using in [7] for Coreutils v6.10. These bugs are not fixed in v6.11. The bug in Row 1 is an integer overflow that leads to memory exhaustion and still exists in the most recent version (v8.25-11799198). The remaining bugs are heap-based buffer overflows in the regular expression component of GNULIB that is used by Coreutils.

Results (Effectiveness). On the benchmark, AFLFAST and KLEE fail to discover one bug in the KLEE benchmark. Moreover, AFLFAST discovered five bugs that KLEE did not find, one of which still existed in the most recent version of Coreutils (Figure 15). The four buffer overflows in ptx that only AFLFAST could detect should be within the search space of KLEE. However, we are unsure whether AFLFAST even has the capability to detect that ptx tries to allocate a negative amount of memory, which together with an integer overflow leads to memory exhaustion. We investigated the failure to detect the bugs within the one hour time budget. For AFLFAST, we attribute the failure to discover the bug in tac to recent changes in KLEE. For AFLFAST, we attribute the failure to discover the bug in md5sum to the difficulty to synthesize the specific sequence of characters required to expose the bug (Figure 14). AFLFAST must generate a file containing a specific sequence of eight bytes and the option -c to expose the bug. We note, however, that AFLFAST can detect the bug with a two hour time budget.

AFLFAST outperforms KLEE in terms of effectiveness within the 1-hour time frame.

Results (Coreutils v8.25). On Coreutils v8.25, AFLFAST discovers eight previously unreported bugs while KLEE discovers only one (Figure 16). We attribute much of this advantage to AFLFAST’s runtime checking while KLEE requires a constraint-based error detection mechanism. KLEE symbolically executes (i.e., interprets) the program’s LLVM intermediate representation and uses constraint solving to determine whether an error could have occurred. Most symbolic-execution-based whitebox fuzzers integrate the error detection directly into the constraint solving process. This embedding restricts the detection to such errors that can be encoded as constraint violations. For instance, it might be difficult to detect (and thus to detect) the misaligned address bug that AFLFAST could find. The error detection is further impaired by the incompleteness of the environment model that underlies the symbolic interpreter. In contrast, AFLFAST as greybox fuzzer executes the compiled binary concretely and reports any generated input that crashes. Runtime checkers, such as ASAN [23], crash the program for executions where the program would not normally crash.

To detect vulnerabilities, KLEE requires a constraint-based vulnerability detection mechanism and presumably completes of constraint encoding and environment modelling while AFLFAST merely executes the program reporting any crashes, e.g., using runtime checking.
6.5 Code Coverage

We compare KLEE and AFLFAST in terms of coverage achieved. Leveraging our earlier experimental infrastructure, we choose the most recent version of GNU Coreutils (v8.25-f1799198) as subjects. The experimental setup as discussed in Section 6.3 remains. We measure statement coverage achieved after one hour of fuzzing by re-executing the generated inputs on the utilities compiled for GCOV.

Results. On average, AFLFAST achieves slightly more code coverage than KLEE (Table 4). While KLEE covers circa 78% of executable lines of code in the average utility, AFLFAST covers about 82%. In total, there are 16k executable statements, out of which AFLFAST covers slightly more than 12k (76%) and KLEE slightly more than 10k (64%). This means that AFLFAST scales better than KLEE and is more effective when covering statements in the larger utilities. However, in our experience (i.e., by checking the statements in some utilities that were covered by one fuzzer or the other) KLEE and AFLFAST have individual strengths and perform best in different circumstances. For instance, KLEE could easily synthesize complex command-line options (e.g., --help) that would be out of reach for AFLFAST running without a dictionary. The results might have looked very different if the majority of options were not just two characters long (e.g., -h).

AFLFAST performs slightly better than KLEE in terms of code coverage. However, a combination might be even more powerful, using their strengths to mitigate each other's weaknesses.

7 Deployment in AFL 2.33B

Together with the publication of the shorter version of this paper, we made AFLFAST available to the public at https://github.com/mboehme/aflfast

The release of AFLFAST as Github repository created considerable interest among practicing cyber-security researchers. Links to the paper as well as to the repository were shared and re-tweeted hundreds of times on Twitter. Members of the security community downloaded our tool and evaluated it on their own, finding a large number of bugs that were previously out of reach. The blogposts they wrote were featured for several hours as TOP-1 entry on Hackernews, a popular forum for tech-savvy readers. Google security awarded USD 2,000 in bug bounties for the bugs found by AFLFAST (and reported in this paper).

Only eight months after its release, the Github repository of AFLFAST received 172 stars and was forked 89 times. Even though the changes to the source code were relatively small, the impact on the security community was immense.

Michal Zalewski, the main developer of AFL, read our paper, studied our implementation, improved AFL according to several observations, and conducted experiments of his own.11 We report on his observations about AFLFAST, the resulting improvements to AFL and the outcome of his experiments. Zalewski observed the following about our implementation:

1) Rather than spending time in the deterministic stage, AFLFAST spends a greater proportion of time on "non-deterministic" fuzzing early in the campaign. In other words, AFLFAST assigns substantially less energy in the beginning of the fuzzing campaign.

2) Most of the cycles that AFLFAST carries out, are in fact very short. This causes the queue to be cycled very rapidly, which in turn causes new retained inputs to be fuzzed almost immediately. In other words, because AFLFAST assigns less energy, it can process the complete queue substantially faster. We say it starts by exploration rather than by exploitation.

Based on these observations, Zalewski implemented into mainline AFL what we call the explore-schedule. Concretely, he reduced the time a seed is being fuzzed by a factor of 20. For the experiments, he calls this new version FidgetyAFL (version 2.33b) – not to be confused with the older versions which implement the exploit-schedule (up to version 2.32b). For the experiments, Zalewski compared AFL, FidgetyAFL, and AFLFAST on four subjects. Instead of the number of paths or number of crashes, Zalewski measures the map density. The map density correlates with the number of edges that have been exercised in the subject's control-flow graph. All fuzzers are started with the same parameters.

- libpng (2 runs of 6h). AFL was consistently but slightly ahead of AFLFAST (~1%). FidgetyAFL outperformed both AFL and AFLFAST by about 10% at the one (1) hour mark but slightly trailed behind at the six (6) hour mark. This is consistent with our observation that the exploit-schedule is quite efficient only for the first few hours—but is later outperformed by other schedules (see Figure 12).

- libjpeg-turbo (2 runs of 6h). AFLFAST outperformed AFL by 15% at the one hour mark and by 5% at the

11. The discussion can be found here: https://groups.google.com/forum/#!topic/afl-users/FOPe62FZUg
six hour mark. FidgetyAFL outperformed AFL by 13% at the one hour mark and by 4% at the six hour mark.

- **zlib** (3 runs of 1h). AFLFAST and AFL perform similarly in the beginning but AFLFAST takes a solid lead of 20% later on. FidgetyAFL and AFLFAST perform similarly throughout.

- **patch** (3 runs of 1h). AFLFAST outperforms AFL very decisively by 50% while FidgetyAFL and AFLFAST perform similarly.

We note that the experiments may not be statistically sound (i.e., only results for two to three fuzzing campaigns of relatively short time are reported). Specifically, it may take some more time for the exponential schedule implemented in AFLFAST to gain traction and outperform the explore schedule implemented in FidgetyAFL (see Figure 12).

Since version 2.33b, AFL implements the explore-schedule. This does not require to record the number of fuzz exercising a certain path, and in our experiments the explore-schedule outperforms all other schedules for the first few hours (see Figure 12). This revised version of AFL was shown to be significantly more efficient than its predecessors. While AFLFAST still compares favourably against the improved version of AFL, the performance difference to AFL 2.33b and newer versions is smaller than we have shown in our experiments.

We also conducted a small experiment where we compared FidgetyAFL and AFLFAST in terms of number of paths discovered on two security-critical programs: OpenSSL [25], a library for secure communication over the internet and LibXML2 [26], a popular XML parser library for C. For OpenSSL, we fuzzed the asml utility and for LibXML2 the xmllint utility. Both were fuzzed without additional command line parameters, without the deterministic stage (i.e., using AFL option -d), for 24 hours, and using only the empty file as initial seed. We conducted ten runs for FidgetyAFL and 10 runs for AFLFAST.

The results are shown in Figure 17. Overall, AFLFAST is more efficient than FidgetyAFL. For LibXML2, FidgetyAFL takes 24 hours to discover the same number of paths that AFLFAST discovers in 9 hours (i.e., about 3× longer). For OpenSSL, FidgetyAFL takes 24 hours to discover the same number of paths that AFLFAST discovers in 19.5 hours.

### Figure 17. Average #paths discovered in LibXML2 and OpenSSL in 10 runs of 24 hours using FidgetyAFL and AFLFAST.

8 **Related Work**

Several techniques [12], [27], [28], [29] have been proposed to increase the efficiency of fuzzing. An important optimization pertains to selecting the seed inputs wisely from a wealth of inputs [28]. Our work makes no assumptions about the existence 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 [29]. 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. [27] 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 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 fuzzers can generate files that stress low-frequency paths. Probabilistic symbolic execution [30] 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 [7], [14], [15], [31]. It can generate the specific values that are needed in order to negate an if-condition and exercise the alternative branch. Taint-based fuzzing [16], [32] 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 [33] leverages an input model to synthesize and “transplant” complete data chunks to exercise so-called critical branches that are only exercised if a certain 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 [2].

Colleagues have combined lightweight black- or greybox fuzzers with symbolic execution-based whitebox fuzzers to get the best of both worlds [8], [34]. For instance, HybridFuzz [34] first runs symbolic execution to generate inputs leading to “frontier nodes” and then passes these inputs to a blackbox fuzzer. In contrast, Driller [9] 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. [35] 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. [36] 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.

Markov chains are often used by meta-heuristic search algorithms to solve optimization problems. A demonstration of the utility of the Markov chain model of CGF is our recent work. Directed greybox fuzzing [37] uses a novel set of power schedules that implement the Simulated Annealing meta-heuristic to minimize the distance to a given set of target locations in the program. This is important for patch testing or directed vulnerability detection where dangerous locations are known. These power schedules assign more energy to seeds that are “closer” to the targets. At compile-time, the control-flow graph is analyzed and the program instrumented, such that at runtime the seed distance can be efficiently computed. Using the Markov chain model of CGF, we cast the reaching of target locations as optimization problem. Other fuzzing objectives may also be encoded.

9 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 took a state-of-the-art greybox fuzzing AFL which keeps track of path identifiers. We enhanced 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.

ACKNOWLEDGMENTS

We thank Michal Zalewski, all members of the AFL community, and the members of Team Codejitsu for the interesting discussions about our research and the independent evaluation of AFLFAST. 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.

REFERENCES

Van-Thuan Pham is a postdoctoral research fellow at NUS, and he is also the Technology Lead of Test1080, a spinoff from NUS which provides an intelligent mobile app testing service based on a patent-pending technology. He is passionate about doing R&D on automated testing technologies to improve the reliability of software systems running on all types of computing devices such as embedded systems, mobile devices, personal computers and servers. During his PhD studies at NUS, under the supervision of Prof Abhik Roychoudhury he conducted research on fuzz testing techniques (including black-box, coverage-based grey-box and symbolic-execution based white-box fuzzing) and applied these techniques to vulnerability detection. 

Abhik Roychoudhury is a Professor of Computer Science at National University of Singapore. His research focuses on software testing and analysis, trust-worthy software construction and software security. He has been an ACM Distinguished Speaker (2013-19). He is currently leading the TSUNAMI center, a large five-year long targeted research effort funded by National Research Foundation in the domain of trust-worthy software. He is also the Lead Principal Investigator of the Singapore Cyber-security Consortium. His research has been funded by various agencies and companies, including the National Research Foundation (NRF), Ministry of Education (MoE), A*STAR, Defense Research and Technology Office (DRTech), DSO National Laboratories, Microsoft and IBM. He has authored a book on "Embedded Systems and Software Validation" published by Elsevier (Morgan Kaufmann) Systems-on-Silicon series in 2009, which has also been officially translated to Chinese by Tsinghua University Press. He has served in various capacities in the program committees and organizing committees of various conferences on software engineering, specifically serving as Program Chair of ACM International Symposium on Software Testing and Analysis (ISSSTA) 2016. He has served as an Editorial Board member of IEEE Transactions on Software Engineering (TSE) from 2014-18. Abhik received his Ph.D. in Computer Science from the State University of New York at Stony Brook in 2000.