Coverage-based Fuzzing as Markov Chain

Marcel Böhme Van-Thuan Pham Abhik Roychoudhury

TSUNAMi Security Research Center, National University of Singapore
{marcel,thuan,abhik}@comp.nus.edu.sg

ABSTRACT

In this paper, we model greybox fuzzing as a systematic exploration of the state space of a Markov chain. The chain specifies the probability that fuzzing an input that exercises path i generates an input that exercises path j. Then, we assign each fuzzed test input an *energy* that controls the amount of fuzz generated at each iteration. We suggest that greybox fuzzing is most efficient if the exploration focuses on the *low-density region* of the stationary distribution and the energy is regulated using a pre-defined *power schedule*. We implemented the exponential schedule as an extension of AFL. In 24 hours, AFL-Fast exposes 3 previously unreported CVEs that are not exposed by AFL and exposes 6 previously unreported CVEs 7x faster than AFL. AFL-Fast yields at least one order of magnitude more unique crashes than AFL.

Keywords

Vulnerability Research; Path Exploration; Efficiency

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 (author of AFL) [1]

Today, most vulnerabilities are found with automated test generation techniques. There has been much debate about the effectiveness of symbolic execution and the efficiency of fuzzing techniques [1, 3]. Fuzzing stresses a system with large amounts of random data in an attempt to make it crash while symbolic execution systematically explores the paths in a program. Fuzzing might exercise only "shallow" paths while symbolic execution can find vulnerabilities that hide deep in the program. Fuzzing is a lightweight technique while symbolic execution requires some heavy machinery for the program analysis and constraint solving. Recently, there have been attempts to combine both techniques and deploy symbolic execution for paths where fuzzing "gets stuck" [21].

CCS'16 October 24–28, 2016, Vienna, Austria
© 2016 ACM. ISBN 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

Can we make fuzzing more effective at path exploration? Many before us have suggested to make symbolic execution faster at path exploration [14, 4, 20, 12]. In this work instead, we make fuzzing, specifically *coverage-based greybox fuzzing* (CGF), better at path exploration. CGF uses lightweight (binary) instrumentation to determine a unique identifier for each program path that is exercised by a generated input. Inputs are generated by slightly mutating the provided or discovered seed inputs. If a generated input exercises a new (and interesting) path, the fuzzer retains the input, otherwise it discards it. The fuzzer mutates the discovered seed inputs in a continuous loop. Increasing the efficiency of greybox fuzzing means to expose significantly more vulnerabilities and achieve more coverage within the same time.

CGF does not require any program analysis. Hence, there is no imprecision in the lifting or modelling of a program's control-flow, of its memory layout, or of any encoding in Satisfiability Modulo Theory (SMT). Without any time spent on program analysis, there is also less concern about the scalability of the approach. CGF is trivially highly parallelizable where the discovered seeds represent the only internal state. AFL is a state of the art coverage-based greybox fuzzer that is behind hundreds of high-impact vulnerability discoveries [23] and has been shown to generate valid JPEGs from seed input that was virtually empty.[1] Increasing the efficiency of greybox fuzzers like AFL has a real and practical impact on vulnerability research.

Fuzzers are often provided with seed inputs that exercise some functionality of the tested program. The hope is that inputs that are generated by slightly mutating the seeds will remain *valid* but still exercise different (buggy) functionalities in the program. For instance, suppose we are testing an image library and we have some example JPEG files. The fuzzer is more likely to exercise paths deep in the program if it generated inputs by slightly mutating the existing JPEG files, than if it generated inputs completely from scratch and at random. We call inputs that are generated by fuzzing input t the *fuzz* of t.

The objective of efficient fuzzing is to exercise a maximum number of paths while generating a minimal number of fuzz. However, there is a natural tendency to exercise the same few paths. For instance, there may be a 90% chance that fuzzing a *valid* JPEG generates an input that exercises a path r which rejects invalid inputs while there may be a 99.999% chance that fuzzing an *invalid* "JPEG" generates another invalid input that exercises r. Informally, we call r a *high-frequency* path because it is exercised by most fuzz.

[1] https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

In this paper, we propose several strategies that allow a CGF to spend more time trying to discover low-frequency paths while spending less time executing high-frequency ones. The results are encouraging, indeed. Our AFL extension *AFL-FAST* discovered 9 vulnerabilities in GNU binutils; eight CVEs were assigned and one CVE request is still pending. *AFL-FAST* 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. Our strategies focus on the time for which an input is fuzzed and the order in which inputs are selected. We suggest that a CGF begins with a small fuzzing time which is steadily increased for low-frequency paths and the most promising paths are chosen next.

A common strategy is to spend a *long period* trying to discover low-frequency paths. For instance, AFL fuzzes an input for almost one minute before moving on to the next input (generating about $80k$ inputs). However, most fuzz exercises the same few high-frequency paths. For instance, we tested the `nm`-tool, and in the first ten minutes half of the inputs that AFL generates (and discards) exercise the same three paths.[2] A stronger focus on low-frequency paths i) allows to waste no time generating even more fuzz that exercise the same few high-frequency paths and ii) it allows to discover more paths per generated input.

A better strategy is to spend only a *small* period exercising high-frequency paths, *too*. Spending a lot of time fuzzing a single input allows to discover "neighbors" exercising low-frequency paths. For instance, if there is a 0.1% chance that fuzzing a valid JPEG generates another valid JPEG, then it should be fuzzed at least $1k$ times. On the other hand, if there is only a 10^{-9} % chance that fuzzing an invalid input generates a valid JPEG then perhaps the time is better spent on more progressive inputs.

So, when to spend more and when less time fuzzing an input? Suppose, we know that by fuzzing input t_i the fuzzer discovers the low-frequency[3] path j with probability p. Then, the fuzzer discovers path j on average after $1/p$ inputs are generated by fuzzing t_i. This is a most exact estimate of the required fuzzing time per input. In practice, however, p is clearly unknown. Moreover, if path i that is exercised by t_i was only recently discovered, it is unknown whether i itself is a low-frequency path. In other words, after retaining a new file `f1.jpg` that was generated by fuzzing a valid JPEG, we would like to spend more time fuzzing `f1.jpg` if it was also a valid JPEG that exercises a low-frequency path than if it was an invalid input that exercises a high-frequency path. Yet, since `f1.jpg` was only recently generated it is unclear whether it belongs to the former or the latter category.

Intuitively, coverage-based greybox fuzzers should monotonically increase the time spent fuzzing input t_i while discounting the frequency with which path i is exercised. The degree of increase is controlled by a schedule. If an input turns out to exercise a high-frequency path, we rest assured that it was not fuzzed too much. If an input turns out to exercise a low-frequency path, it is fuzzed more next time. To establish early whether a path is high- or low-frequency, the fuzzer prioritizes inputs that have not been fuzzed very often and inputs exercising low-frequency paths. Otherwise, the selection of only favourite inputs and the evaluation of quality remains in place.

[2]See discussion of Figure 1.

[3]Still, we contend with *some* definition of "low-frequency". Note that frequency itself depends on the fuzzing strategy.

To put our strategies on formal foundations, we model coverage-based greybox fuzzing as the efficient exploration of the state space of a Markov chain. A famous use of Markov chains is for the computation of the Google PageRank [5]. The PageRank value of a page reflects the chance that the random surfer will land on that page by clicking a sequence of links. Pages are the states while the links are the edges that are assigned equal propability. Given page i with one link to page j and ten total links, the probability to reach j from i is one in ten. An ensemble of random surfers that start at a page and follow links from page to page will visit some pages more often and others less often. PageRank considers "high-frequency" pages more important and assigns them a higher score. Formally, the PageRank is an approximation of the *stationary distribution* of the Markov chain. In this context, CGFs may be compared to crawlers with the objective to discover, using a minimal number of clicks, a maximal number of pages – notably including those with a very low PageRank.

MC Model. We model the probability that fuzzing an input which exercises path i generates an input which exercises path j as transition probabilities in a Markov chain. Path frequency can be formally defined by means of the stationary distribution. We assign each state i an *energy* that specifies the number of inputs that are generated by fuzzing t_i next. A *power schedule* regulates the energy that the fuzzer invests into a state. The *objective* of efficient greybox fuzzing is to "discover" an undiscovered state in a low-density region while expending the least amount of total energy.

Evaluation. We evaluated *AFL-Fast* and several schedules plus search strategies on the binutils.[4] An exponential schedule outperforms the other schedules while our search strategies turn out to be effective. In eight runs of six hours, *AFL-Fast* with an exponential schedule found an average of more than one order of magnitude more unique crashes than AFL for the tools `nm` and `cxxfilt`; it found crashing inputs for `objdump` where AFL did not expose any crashes at all. In eight runs of 24 hours, *AFL-Fast* found 6 vulnerabilities in `nm` 7x faster than AFL and exposed 3 vulnerabilities that were not exposed by AFL. *AFL-Fast* 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 DARPA Cyber Grand Challenge establishes similar results. On average, *AFL-FAST* exposes an error 19 times faster than AFL and also exposes 7 errors that are not found by AFL, at all.

Contributions. At a broad level, our contributions can be viewed as bringing in the power of smart path coverage to greybox fuzzing. Symbolic execution allows to systematically enumerate program paths by encoding a path as a logical formula and negating portions of it. Unfortunately, fuzzers trade this systematic path coverage for scalability and tend to visit certain paths with high frequency. The main conceptual contribution of our work is to smartly control how much time is spent in a path – thereby veering the search towards low frequency paths where vulnerabilities may lurk. Technically, we achieve this enhanced path coverage by visualizing greybox fuzzing as a Markov chain, and then developing exploration strategies which forces the fuzzing into states representing low-frequency paths.

[4]https://www.gnu.org/software/binutils/

2. BACKGROUND

2.1 Coverage-based Greybox Fuzzing

Fuzz – an automated random testing tool was first developed by Miller et al. [15] 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 [25, 26, 27, 24]. There are three major categories depending on the degree of leverage of internal program structure: *black-box fuzzing* only requires the program to execute [25, 26, 28] while *white-box fuzzing* [6, 11, 8, 9] requires full access to the source code, for instance, to construct the control-flow graph. *Greybox fuzzing* is situated inbetween both 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) [24] 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. The array `shared_mem[]` array 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 test inputs are choosen in a continuous loop until a timeout is reached or the fuzzing is aborted (line 7). AFL classifies a test input 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 inputs.

For each input t, the CGF determines the time to fuzz t and more specifically the number of inputs generated by fuzzing t (line 8). AFL's implementation of perf_score depends on 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/insertion strategies to generate new inputs.[5]

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

Algorithm 1 Coverage-based Greybox Fuzzing

```
Input: Seed Inputs $S$
 1: $T_✗ = \emptyset$
 2: $T = S$
 3: if $T = \emptyset$ then
 4:     add empty file to $T$
 5: end if
 6: repeat
 7:     $t$ = chooseNext($T$)
 8:     $p$ = perf_score($t$)
 9:     for $i$ from 1 to $p$ do
10:         $t'$ = mutate_input($t$)
11:         if $t'$ crashes then
12:             add $t'$ to $T_✗$
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 $T_✗$
```

If the generated input t' crashes the program, it is added to the set $T_✗$ of crashing inputs (line 12). If t' is considered to be progressive, 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 inputs on the queue. More specifically, t' is interesting if t' executes a path where transition b is exercised n times and for all inputs t'' on the queue that exercise b for m times there does not exist an exponent $x \in \mathbb{N}$ such that $2^x \leq n < 2^{x+1}$ *and* $2^x \leq m < 2^{x+1}$. AFL uses this "bucketing" to address path explosion [21]. Intuitively, AFL retains inputs that execute 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.

While it is quite simple to instrument the source code, a CGF technique, like AFL, does not necessarily require access to the source code. AFL-DynInst[6] is a binary instrumentation tool that injects the instrumentation shown in Listing 1 directly into the execution binary. QEMU [2] is a machine emulator that instruments the binary at runtime.

2.2 Markov Chain

A Markov chain is a stochastic process that transitions from one state to another [16]. 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 $\mathbb{P}(X_0 = i)$.

[6]https://github.com/vrtadmin/moflow/tree/master/afl-dyninst

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 s, 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

$$\begin{aligned} 0 &\le \pi_j \le 1 \\ 1 &= \sum_{i \in S} \pi_i \\ \pi_j &= \sum_{i \in S} \pi_i p_{ij} \end{aligned}$$

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 in the internet. Brin and Page [5] 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 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 test input which exercises some program path i generates an input which exercises path j as transition probabilities p_{ij} in a Markov chain. This allows us to discuss the objective of greybox fuzzing as the efficient exploration of the state space, specifically of the low-density region of the stationary distribution of a Markov chain. Notice the difference to a web crawler which is more interested in indexing the important pages in the high-density region 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 Fuzzing as Markov Chain

Time-inhomogeneous model. To motivate our model of greybox fuzzing, we start with a simplified one. Suppose, after providing the fuzzer with an initial seed input t_0 that exercises path 0, the fuzzer continuously explores path $i+1$ by randomly mutating the previous input t_i which exercises path i. 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. If a different input t'_i was generated that also exercises path i, the probability 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 for greybox fuzzing. The *state space* of the Markov chain is defined by the discovered paths and their immediate neighbors. Given inputs 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 $t \in T$.[7] 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 $t_i \in T$, then p_{ij} is the probability that randomly mutating t_i generates an input that exercises the path j. Else if path i is an undiscovered path that is not exercised by some $t \in T$, 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 assume that generating t_j from t_i is as likely as generating t_i from t_j and that until the undiscovered path j is exercised it 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 discover new 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 and the test input t is added to the circular queue T.

Long tails. In our experiments, we observe several notable properties of such Markov chains. For one, the stationary distribution has a very large number of very-low-density regions and a very 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 $1k$ up 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

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

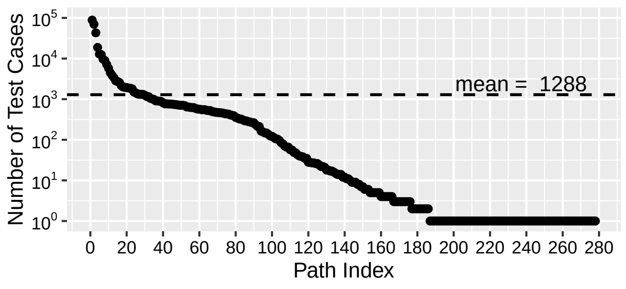


Figure 1: #Fuzz exercising a path (on a log-scale) after running AFL for 10 minutes on the nm-tool.

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 input exercising a low-frequency path or not, or whether the walker started with a valid or an invalid input.

Benefits. The Markov chain model of greybox fuzzing has several benefits. For example, it opens greybox fuzzing as a tool 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 [13] that offer guarantees on the convergence to the true value. In the context of vulnerabilty research, the Markov chain model allows to cast the objective of fuzzing as an efficient exploration of the state space of a Markov chain.

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!”.

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

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*) to randomly mutate an input $s = \langle c_0, c_1, c_2, c_3 \rangle$. *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.

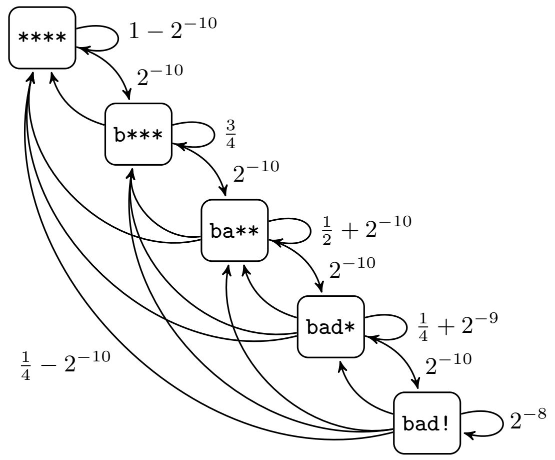


Figure 2: Markov chain for motivating example

Figure 2 represents the simplified transition matrix p_{ij} as a state diagram.[8] 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. A long time will pass until the walker reaches the state `bad!`.

3.3 Exploring the Markov Chain

A *greybox fuzzer* is an ensemble of random walkers in the Markov chain. There is one walker for each input t in the circular queue T. The *objective* is to discover an undiscovered and interesting path that is not exercised by any $t \in T$ using the least number of steps. Conceptually, all walkers

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

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 $p = \text{perf\_score}(t)$ in Algorithm 1. Usually, $p < M$ where $M \in \mathbb{N}$ gives an upper bound on the number of generated inputs. In AFL, $M \approx 160k$.

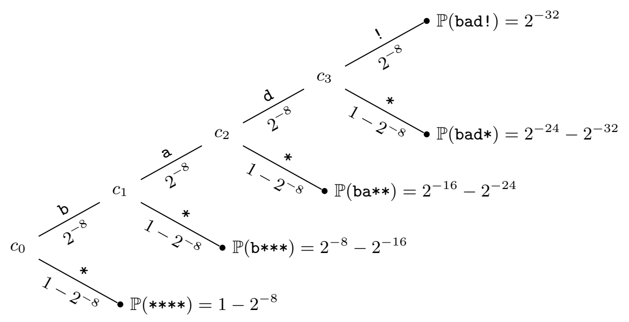
AFL as greybox fuzzer. On our system, AFL takes about one minute to fuzz a single test input $t \in T$. After an initial burn-in, each time t is chosen from the circular queue T, the fuzzing time is usually the same. In other words, the value returned by AFL's implementation of $\text{perf\_score}(t)$ is fairly high and fairly constant. To decide on the next test input to fuzz, AFL first marks as "favourite" input one which is the fastest and smallest executing any given basic block transition and then chooses as next input that favourite input which follows the last fuzzed input in the circular queue T. In other words, AFL's implementation of chooseNext skips some rather big and slow inputs in the circular queue.

Next, we illustrate the behavior of a greybox fuzzer if the number of generated test inputs p is fairly high and constant.

#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.

Example. Let the number of generated inputs per fuzzing round be $p = 2^{16} = 64k$. Since most 4-character words do not start with 'b', the first path that is exercised is likely Path 0. After 2^{16} inputs have been generated by fuzzing the initial test input, several inputs are expected to begin with the letter 'b'. So, one test input is retained that exercises Path 1. After another 2^{16} inputs have been generated by fuzzing the retained test input, at least one input is expected to exercise Path 2 and is retained. Figure 3 shows how the procedure continues. After $256k$ test inputs were generated from the four inputs that were retained for each path, the crashing input is found. As shown in Figure 4, the random generation of the same string would require four *orders of magnitute* more test inputs on average, $2^{32} = 4G$.



$\mathbb{P}(\texttt{bad!}) = 2^{-32}$
!
2^{-8}
c_3
d
2^{-8}
*
$1-2^{-8}$
$\mathbb{P}(\texttt{bad*}) = 2^{-24} - 2^{-32}$
c_2
a
2^{-8}
*
$1-2^{-8}$
$\mathbb{P}(\texttt{ba**}) = 2^{-16} - 2^{-24}$
c_1
b
2^{-8}
*
$1-2^{-8}$
$\mathbb{P}(\texttt{b***}) = 2^{-8} - 2^{-16}$
c_0
*
$1-2^{-8}$
$\mathbb{P}(\texttt{****}) = 1 - 2^{-8}$

Figure 4: The crash is found after $2^{32} = 4G$ inputs were choosen uniformly at random from the set of all 4-character words where c_n is the n-th character.

4. BOOSTING GREYBOX FUZZING

Greybox fuzzers spend *too much time* generating inputs that exercise paths in high-density regions. Paths in high-density regions are often exercised by invalid inputs. More intesting are paths in low-density regions that trigger different program behaviors. So then, a greybox fuzzer should spend *sufficient time* generating inputs that exercise paths in a low-density region. The crux of the matter is that it is unknow a-priori whether fuzzing an input that exercises a newly discovered path leads to the first or the second.

Energy. We define the *energy* $p(i)$ of a discovered state $i \in S^+$ in the Markov chain as the number of test inputs that are generated by fuzzing test input t_i when t_i is next chosen from the circular queue. Let X_{ij} be the random variable that describes the energy that is required for a discovered state $i \in S^+$ until the greybox fuzzer explores an undiscovered state $j \in S^-$ where the probability p_{ij} to transition from i to j is greater than 0. Then,

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

A *power schedule* regulates the energy that the fuzzer invests into a state i (or the fuzzing time for i) when t_i is chosen next from the circular queue. Typical greybox fuzzers implement constant power schedules. In this work, we propose and evaluate several monotonous schedules.

Objective. The most efficient greybox fuzzer explores an undiscovered state in a *low-density region* while expending *the least amount of 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 practice, it is difficult to get a good estimate of the expected number of generated test inputs $\mathbb{E}[X_{ij}]$ required to discover path $j \in S^-$ or whether j is in a low-density region. After all, j is still undiscovered. Hence, greybox fuzzers usually implement several heuristics.

Today, the energy $p(i)$ that greybox fuzzers assign to a state i is often large and fairly constant. On our system, AFL takes up to one minute to fuzz a single test input. AFL can be started with seed inputs that exercise paths in low-density regions (like valid JPEG files if we are testing a image library). If a lot of energy is invested, the low-density neighborhood of the seeds can be explored more thoroughly. This does also address the rapid mixing time of the Markov chain. However, if an input is chosen that exercises a path in a high-density region, the fuzzer is better off investing less energy or no energy at all. In simple terms, the fuzzer should spend more energy fuzzing valid JPEG files (exercising paths in a low-density region) than fuzzing invalid JPEG files (exercising paths in a high-density region).

In this paper, we propose to regulate the energy that is assigned to a state using a power schedule that starts with low power and increases the energy in a monotonous manner. Once a new state i is discovered, its energy $p(i)$ is low. Only a small number of inputs are generated by fuzzing t_i next time it is chosen from the queue. 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. When the energy $p(i)$ reaches a constant $M \in \mathbb{N}$, the maximum number test inputs are generated. Intuitively, after the general neighborhood is explored and it is established that i is in a low-density domain, 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

The required energy for a discovered state is regulated by a *power schedule*. In general, the number of inputs $p(i)$ generated by fuzzing input $t_i \in T$ which exercises path i is a function of a) the number of times $s(i)$ that t_i has previously been choosen from the circular queue and b) the number of generated inputs $f(i)$ exercising i. Note that $f(i)$ also counts the number of rejected inputs and the number of those inputs generated by fuzzing *any* retained test input $t \in T$. We call the inputs generated by fuzzing t_i also the *fuzz* of t_i. We discuss and evaluate several power schedules.

The **exploitation-based constant schedule** (EXPLOIT) is implemented by most greybox fuzzers. After some burn-in, the fuzzing time for an input t_i is kept 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

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

where $\alpha(i)$ 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 with a fairly constant but also fairly low fuzzing time. The energy $p(i)$ for state i is computed as

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

where $\alpha(i)/\beta$ maintaints the fuzzer's 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

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

where $\beta > 1$ 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 test inputs 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 test 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 test inputs on average ($4G$) while $\mathcal{F}_1$ without a power schedule where the number of fuzz is fixed at $M = 2^{16}$ 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 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

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

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 moer 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 proportional to the amount of fuzz $f(i)$ that exercises path i.

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

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

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

4.2 Search Strategies

An efficient coverage-based greybox fuzzer prioritizes inputs that have not been fuzzed very often and inputs that exercise low-frequency paths.

Prioritize small $s(i)$. An efficient fuzzer chooses as next input $t_i \in T$ such that the number of times $s(i)$ that an input has been chosen from the queue before is minimal. If there are several such inputs, it chooses the first. However, the greybox 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 $f(i)$. An efficient fuzzer chooses as next input $t_i \in T$ such that the number of fuzz $f(i)$ that exercises path i is minimal. Again, if there are several such inputs, it chooses the first. 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. EXPERIMENTAL SETUP

5.1 Implementation

AFL is a coverage-based greybox fuzzer that collects information on the basic block transitions that are exercised by an input. The binary's source code is not required [2, 22]. 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 PERF_SCORE and CHOOSE_NEXT 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 *AFL-FAST*.

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. *AFL-FAST* 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. *AFL-FAST* 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 $f(i)$.

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.

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.

5.2 Infrastructure

We conducted our experiments on a 64-bit machine with 40 cores each running at 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 simultanously, 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.

6. BINUTILS CASE STUDY

6.1 Vulnerabilities

We chose 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 `cxxfilt`. We zoom into some results by discussing one binutils tool (i.e., `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 Fig. 6).

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)

Figure 6: CVE-IDs and Exploitation Type

All found vulnerabilities were reported to the maintainers. We submitted a patch and informed the security community via the *ossecurity* mailing list.[9] *Eight (8) CVEs* have been assigned while one request is pending. Most patches have been accepted while others are 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 to reverse engineer the binary.

6.2 General Results

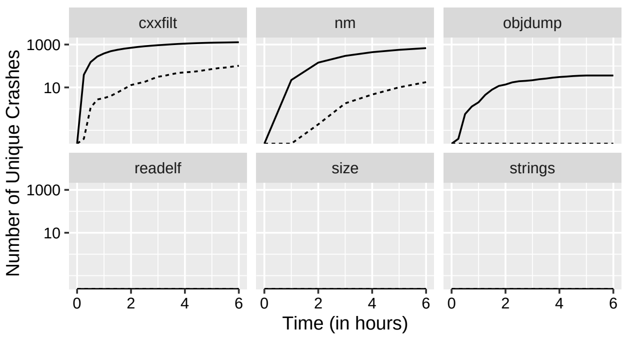


Figure 7: #Crashes over time (on a log-scale) for AFL-Fast (solid line) vs. AFL (dashed line)

Crashes over time. After 6h, *AFL-Fast* found one and two *orders of magnitude* more unique crashes than AFL in `cxxfilt` and `nm`, respectively.[10] *AFL-Fast* 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 7. In what follows, we investigate the unique crashes generated for `nm` with a 24 hour budget in more details.

Vulnerabilities in `nm`. On average, *AFL-Fast* 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. *AFL-Fast* exposes all vulnerabilities in 2h17m, on average while AFL would require more than 12h30m. The first three rows of Figure 8 show the results for the vulnerabilities in the `nm` tool in more details. Each facet compares *AFL-Fast* 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, *AFL-Fast* consistently and significantly outperforms classic AFL. The average time to first exposure is shown in Figure 9. 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`. *AFL-Fast* finds two buffer overflows seven (7) times faster than AFL. *AFL-Fast* 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 6 shows more details. Again, our extension consistently outperforms the classic version of AFL.

[9]http://www.openwall.com/lists/oss-security/2016/05/05/3
[10]Notice the logarithmic scale in Figure 7.

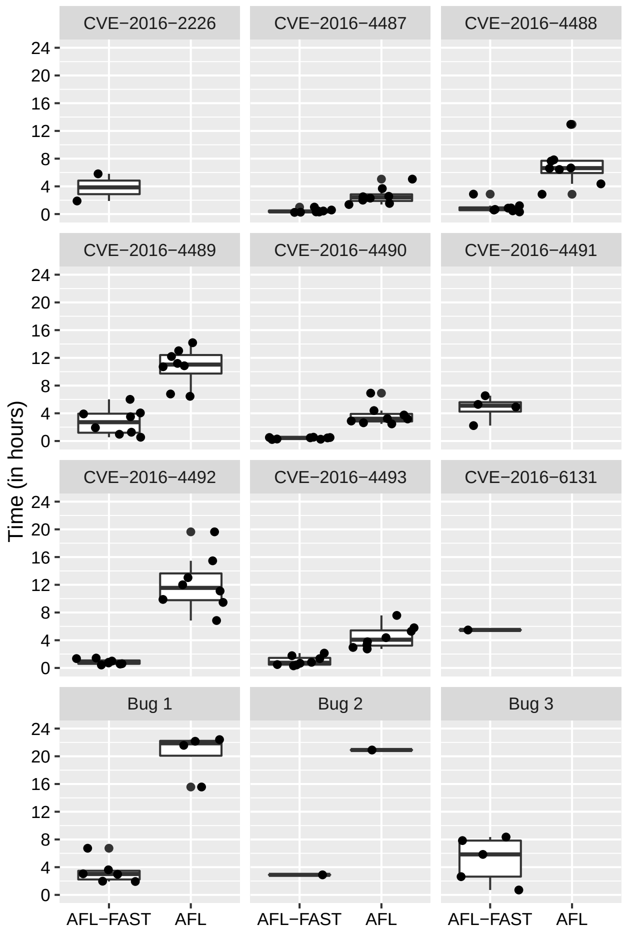


Figure 8: Time to expose the vulnerability.

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 9: Time to expose the vulnerability.

Independent Evaluation. We note that our collaborators, Team Codejitsu at the DARPA Cyber Grand Challenge (CGC), evaluated both AFL and *AFL-Fast* on all 150 benchmark programs that are provided as part of the CGC. On these binaries, *AFL-Fast* exposes errors 19x faster than AFL, on average. In one run, AFL exposed four errors that are not exposed by our extension. However, AFL-Fast exposed seven errors that are not exposed by AFL. A thorough discussion and reflection of the CGC experience could potentially be conducted in the future. This paper is not focused on CGC challenges and the very many mechanisms to tackle them. Instead we focus on the *idea* of boosting greybox fuzzing using Markov chains, and the efficacy of the idea as evidenced by comparison with state-of-the-art fuzzers such as AFL.

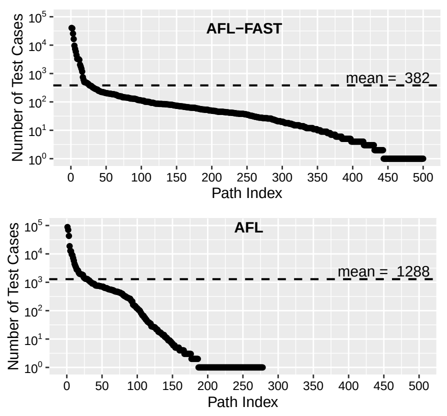


Figure 10: #Fuzz exercising a path (on a log-scale) after running AFL for 10 minutes on the `nm`-tool.

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 10, indeed our heuristics generate more fuzz for low-frequency paths and less fuzz for high-frequency paths. In 10 minutes, *AFL-Fast* discovered twice as many paths as AFL. For *AFL-Fast* 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 *AFL-Fast*. This clearly demonstrates the effectiveness of our heuristics in exploring a maximal number of (low-frequency) paths while expending minimum energy.

6.3 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, *AFL-Fast* implements an exponential schedule.

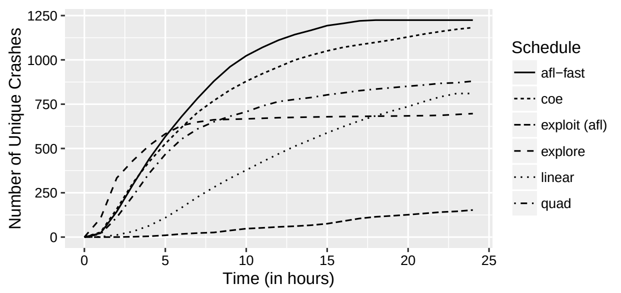


Figure 11: #Crashes over Time (Schedules).

Results. The exponential schedule that is implemented in *AFL-Fast* outperforms all other schedules. The cut-off exponential schedule (coe) performs only slightly worse than *AFL-Fast*. 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 *AFL-Fast* but at the end of the 24 hour budget it performs comparably to the other two (linear and explore).

6.4 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.[11] Without Strategy 1, *AFL-Fast* (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 *AFL-Fast* (like AFL) chooses the next input from the queue that is marked as favourite. All strategies are run with the exponential power schedule.

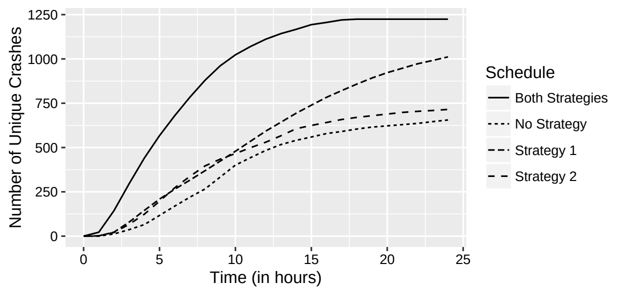


Figure 12: #Crashes over Time (Search Strategies).

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 *AFL-Fast* without any strategies (comparable to AFL but with exponential power schedule). However, after 24 hours, *AFL-Fast* with both strategies exposes almost twice as many unique crashes as *AFL-Fast* with no strategy or with only strategy 1.

[11]For more details see Section 5.1.

6.5 Result Summary

We evaluated *AFL-Fast* and several schedules plus search strategies on the GNU binutils.[12] The exponential schedule outperforms all other schedules while our search strategies turn out to be effective. In eight runs of six hours, *AFL-Fast* with an exponential schedule found an average of more than one order of magnitude more unique crashes than AFL for the tools `nm` and `cxxfilt`; it found crashing inputs for `objdump` where AFL did not expose any crashes at all. In eight runs of 24 hours, *AFL-Fast* found 6 vulnerabilities in `nm` 7x faster than AFL and exposed 3 vulnerabilities that were not exposed by AFL. *AFL-Fast* 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, *AFL-FAST* exposes an error 19 times faster than AFL and also exposes 7 errors that are not found by AFL, at all.

7. RELATED WORK

Several techniques [30, 18, 7, 24] have been proposed to increase the efficiency of automated fuzzing. Seed selection techniques [18] allow to choose the seed inputs wisely from a wealth of inputs. We do not make any assumption about existing seed inputs and seeded our experiments with a single empty file. However, both AFL and *AFL-Fast* would clearly benefit from a smart seed selection if many seed files are available. Program-adaptive mutational fuzzing [7] uses whitebox analysis to detect dependencies among the bit positions of an input, and then uses this dependency relation to fuzz dependent positions only together. In our work, we retain the mutation ratio and do not require any whitebox analysis. Woo et al. [30] recognize the exploration-exploitation trade-off between fuzzing an input for a shorter or for a longer amount of time and model blackbox fuzzing as a multi-armed bandit problem. However, the decision is based on whether or not the input has exposed a (unique) crash in any previous fuzzing iteration. In contrast, we leverage information on the increase in path coverage and the fuzz exercising a path. This allows us to model coverage-based greybox fuzzing as an efficient exploration of the state space of a Markov chain and specify its objective as exploring a maximal number of states using a smallest amount of generated inputs with a focus on low-density regions. To the best of our knowledge, none of these works report on results exceeding a factor of two in terms of time to exposure.

Probabilitistic Symbolic Execution (PSE) [10] may be used to guide the test generation towards low-frequency paths. PSE combines symbolic execution and model counting to compute the probability that an input that is randomly chosen from the program's input space exercises a given path. Fuzzing and symbolic execution have been combined before [21, 17] where fuzzing helps to overcome the challenges of symbolic execution and vice versa. For instance, HybridFuzz first runs symbolic execution to generate inputs leading to "frontier nodes" and then passes these inputs to a black-box fuzzer that applies mutation strategies. In contrast, Driller [21] begins with blackbox fuzzing and only seeks help from symbolic execution when it gets stuck, for instance, to generate a magic number to bypass a program check.

[12]https://www.gnu.org/software/binutils/

A challenge of greybox fuzzing is the generation of specific values, like magic numbers, checksums. or offset values. Symbolic-execution based whitebox fuzzing allows to substitute certain input bytes that impact the outcome of a branch with symbolic variables and employ symbolic execution to negate those branches [6, 11, 19, 8]. Taint-based whitebox fuzzing [9, 29] is a directed technique that allows to reach certain (dangerous) targets in the program. It exploits taint analysis to localize parts of the input which should be marked symbolic. For instance, it marks input portions which can control arguments of executed, critical system calls as symbolic. Checksum-aware whitebox fuzzing [29] is a simple yet effective idea to fuzz a program with checksum validations. It attemps to identify checksum checks and circumvent them during whitebox fuzzing. Once the test inputs are generated, the checksums are repaired by re-computing them from input data.

8. CONCLUSION

While symbolic execution based testing 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. *AFL-Fast*, our extension of AFL exposes an order of magnitude more unique crashes than AFL in the same time budget. Moreover, *AFL-Fast* can expose several bugs and vulnerabilities that AFL cannot find. Other vulnerabilities *AFL-Fast* exposes substantially earlier than AFL.

More importantly, we provide an explanation of our enhanced effectiveness in the form of visualizing greybox fuzzing as the exploration of the state space of a Markov chain. Like a web crawler that executes a sequence of clicks on web pages in a vast network of pages, a Coverage-based Greybox Fuzzer (CGF) exercises a sequence of paths by fuzzing the input exercising one path until generating an input that exercises another path. However, unlike the web crawler that seeks to index the most important, high-frequency web pages, the CGF seeks to explores as many paths as possible, notably including those (low-frequency) paths that are significantly less likely to be reached – preferably with the smallest number of inputs generated. Unlike the web crawler that follows all links on the same page with equal probability, the CGF has no information how many inputs need to be generated to reach a neighboring path. We have devised several strategies to force the CGF to explore more states in the Markov chain that are hidden in a low-density region and generate less inputs for states in a high-density region.

9. 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.

10. REFERENCES

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

[2] 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.

[3] M. Böhme and S. Paul. On the efficiency of automated testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 632–642, 2014.

[4] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08, 2008.

[5] 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.

[6] 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.

[7] 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.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 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] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, 2013.

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

[14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, 2012.

[15] 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.

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

[17] 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.

[18] 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.

[19] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, 2008.

[20] M. Staats and C. Pǎsǎreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 183–194, 2010.

[21] 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.

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

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

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

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

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

[27] Tool. Suley Fuzzer. https://github.com/OpenRCE/sulley. 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 &#38; Communications Security*, CCS '13, pages 511–522, 2013.