PerfFuzz: Automatically Generating Pathological Inputs

Caroline Lemieux
University of California, Berkeley, USA
clemieux@cs.berkeley.edu

Rohan Padhye
University of California, Berkeley, USA
rohanpadhye@cs.berkeley.edu

Koushik Sen
University of California, Berkeley, USA
ksen@cs.berkeley.edu

Dawn Song
University of California, Berkeley, USA
dawnsong@cs.berkeley.edu

ABSTRACT

Performance problems in software can arise unexpectedly when programs are provided with inputs that exhibit worst-case behavior. A large body of work has focused on diagnosing such problems via statistical profiling techniques. But how does one find these inputs in the first place? We present PerfFuzz, a method to automatically generate inputs that exercise pathological behavior across program locations, without any domain knowledge. PerfFuzz generates inputs via feedback-directed mutational fuzzing. Unlike previous approaches that attempt to maximize only a scalar characteristic such as the total execution path length, PerfFuzz uses multi-dimensional feedback and independently maximizes execution counts for all program locations. This enables PerfFuzz to (1) find a variety of inputs that exercise distinct hot spots in a program and (2) generate inputs with higher total execution path length than previous approaches by escaping local maxima. PerfFuzz is also effective at generating inputs that demonstrate algorithmic complexity vulnerabilities. We implement PerfFuzz on top of AFL, a popular coverage-guided fuzzing tool, and evaluate PerfFuzz on four real-world C programs typically used in the fuzzing literature. We find that PerfFuzz outperforms prior work by generating inputs that exercise the most-hit program branch 5× to 69× times more, and result in 1.9× to 24.7× longer total execution paths.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Software performance*;

KEYWORDS

fuzz testing, performance, algorithmic complexity, worst-case

ACM Reference Format:
Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213874

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213874

1 INTRODUCTION

Performance problems in software are notoriously difficult to detect and fix [39]. Unexpected performance issues can lead to serious project failures and create troublesome security issues. For example, a well-known class of Denial-of-Service (DoS) attacks target algorithmic complexity vulnerabilities [1–3, 5, 23] which cause a running program to exhaust computational resources when presented with worst-case inputs.

A large body of research has focused on diagnosing performance problems by observing or statistically analyzing dynamically collected performance profiles [12, 32, 44, 45, 54]. Almost all of these techniques assume the availability of test inputs with which to execute the candidate program for performance profiling. But where do these inputs come from? The most commonly chosen sources include (1) specially hand-crafted performance tests [43, 45], (2) standardized benchmark suites [12, 13, 22], (3) inputs that are commonly encountered in normal program usage (sometimes called *representative workloads*) [31, 63], or (4) inputs sent by users experiencing performance problems [54]. These sources of inputs either stress only average-case behavior, are subject to human bias and error, or can only be obtained when the damage is already done.

A particular class of inputs which would be useful to developers in alleviating these problems are pathological inputs. *Pathological inputs* are those inputs which exhibit worst-case algorithmic complexity in different components of the program. For example, a program may use data structures such as hash tables and sorting algorithms such as quicksort. Pathological inputs would be those which, when executed, lead to many collisions in the hash table or many swaps in the sorting routine. Such pathological inputs can be identified as those which, given a fixed input length, maximize the execution count of a particular program component.

We present PerfFuzz, a method to automatically generate pathological inputs without any domain knowledge about the program. PerfFuzz generates inputs via *feedback-directed mutational fuzzing*. *Fuzz testing*, in which a program is bombarded with many randomly generated inputs, has been very successful in finding security vulnerabilities and correctness bugs [15, 27, 37, 38, 47, 51, 58, 60]. State-of-the-art fuzzing engines perform *feedback-directed mutational fuzzing* [4, 19, 28, 51, 60]: new inputs are generated by mutating a previously saved input, and new inputs are saved for future mutation if they execute a new program location (i.e. they increase code coverage). The *key idea* in PerfFuzz is to associate each program location to an input that exercises that location the most. Inputs that exercise some program location more than any previous input are saved and prioritized for subsequent mutation. This enables

PerfFuzz to find a variety of inputs that exercise distinct *hot spots* in a program, i.e., program locations that are frequently executed.

We evaluate the ability of PerfFuzz to find hot spots in four real-world C programs commonly used in the fuzzing literature. The inputs generated by PerfFuzz exercise the most-frequently executed program branch 2×–39× times more often than the inputs generated by conventional coverage-guided fuzzing. We also compare PerfFuzz with SlowFuzz [49], a recently-published work on discovering algorithmic complexity vulnerabilities. PerfFuzz outperforms SlowFuzz in discovering inputs exercising worst-case algorithmic complexity in micro-benchmarks. PerfFuzz is also better at generating pathological inputs in macro-benchmarks, finding inputs that exercise the most-frequently executed program branch 5×–69× times more and have 1.9×–24.7× longer execution paths. Unlike SlowFuzz, which tries to maximize only the total execution path length, PerfFuzz uses multi-dimensional feedback and independently maximizes the number of times each program location is executed. We believe the performance response of a program is not necessarily a convex function of its input characteristics. Thus, the multi-dimensional objective may help PerfFuzz escape local maxima, which explains its better performance.

To summarize, this paper makes the following contributions:

(1) We present PerfFuzz, an algorithm for generating inputs that exercise pathological behavior in various program components using feedback-directed mutational fuzzing. Like fuzz testing tools such as AFL, PerfFuzz is input-format agnostic. We release PerfFuzz as an open-source tool[1].
(2) We empirically evaluate the efficacy of PerfFuzz in generating pathological inputs with that of AFL, a conventional coverage-guided fuzzing tool.
(3) We empirically evaluate the efficacy of PerfFuzz in generating pathological inputs with that of SlowFuzz, a similar feedback-directed fuzzing tool designed to find algorithmic complexity vulnerabilities.
(4) We perform a manual analysis of the hot spots discovered by PerfFuzz and describe the insights we gained as to how varying input features affect the performance of different program components.

2 OVERVIEW

2.1 A Motivating Example

The C program in Figure 1 is a simplified version of `wf` [6], a simple word frequency counting tool that is packaged in the Fedora 27 RPM repository. The main program driver (omitted from the figure for brevity) takes as input a string, splits the string into words at whitespaces, and counts how many times each word occurs in the input. To map words to integer counts, the program uses a simple hashtable (defined at Line 11) with a fixed number of buckets. Each bucket is a linked list of entries holding counts for distinct words that hash to the same bucket. As each word is scanned from the input, the program invokes the `add_word` function (Lines 22–40). This function first computes a hash value for that word—implemented in `compute_hash` (Lines 14–20)—and then attempts to find an existing entry for that word (Lines 28–37). If such an entry is found, its count is incremented (Line 31). Otherwise, a new entry is created with a count of 1 (Line 39).

When this program is used to compute word frequencies for an input containing English text, the program does not exhibit any performance bottlenecks. This is because English text usually contains words of short length (about 5 characters on average) and the number of distinct words is not very large (less than 10,000 in a typical novel). However, there are at least two performance bottlenecks that can be exposed by pathological inputs.

First, if the input contains very long words (e.g., nucleic acid sequences, a common genomics application), the program will spend most of its time in the `compute_hash` function. This is because the `compute_hash` function iterates over each character in the word irrespective of its length. For most applications, it is sufficient to compute a hash based on a bounded subset of the input, such as a prefix of up to 10 characters.

Second, if the input contains many distinct words (e.g., e-mail addresses from a server log), the frequency of hash collisions in the fixed-size hashtable increases dramatically. For such an input, the program will spend most of its time in the function `add_word`, traversing the linked list of entries in the loop at lines 28–37. In the worst-case, the run-time of `wf` increases quadratically with the number of words. This bottleneck can be alleviated by replacing the linked list with a balanced binary search tree whenever the number of entries in a bucket becomes very large.

Now, how does the developer of this program identify these performance bottlenecks? If the inputs that exercised the behaviors outlined above were available, then they could run the program through a standard profiling tool such as GProf [32] or Valgrind [44] and observe the source locations where the program spends most of its time. They could also use a statistical debugging tool [54] to compare runs of inputs that take a long time to process versus inputs that are processed quickly. Alternatively, they could use an algorithmic profiling tool [63] to estimate the run-time complexity by varying the size of pathological inputs. But how does the developer acquire such inputs in the first place? Our performance fuzzing technique addresses exactly this concern.

2.2 Performance Fuzzing

Our goal is to generate inputs that independently maximize the execution count of each edge in the control-flow graph (CFG) of a program. We assume that we have one or more *seed inputs* to start with. These seeds are test inputs designed for verifying functional correctness of the program, and need not expose worst-case behavior. In our experiments, we use at most 4 seeds, but usually only 1. In the absence of such seeds, we can also simply start with arbitrary inputs such as an empty string or randomly generated sequences. The basic outline of our input-generation algorithm, called PerfFuzz, is as follows:

(1) Initialize a set of inputs, called the *parent inputs*, with the given *seed inputs*.
(2) Pick an input from the parent inputs that maximize the execution count for some CFG edge.
(3) From the chosen parent input, generate many more inputs, called *child* inputs, by performing one or more *random mutations*. These mutations include randomly flipping input

[1] https://github.com/carolemieux/perffuzz

```
1  // Hash-map entry;  also a linked list node,
2  // to resolve hash collisions
3  typedef struct entry_t {
4    char* key;
5    int value;
6    struct entry_t* next;
7  } entry;
8
9  // Fixed-size table of hash-map entries.
10 const int TABLE_SIZE = 1001;
11 entry* hashtable[TABLE_SIZE] = {0};
12
13 // Computes a hash value for a word.
14 unsigned int compute_hash(char* str) {
15   unsigned int hash = 0;
16   for (char* p = str; *p != '\0'; p++) {
17     hash = 31 * hash + (*p);
18   }
19   return hash % TABLE_SIZE;
20 }
21 // Increments word count in the hash-map.
22 void add_word(char* word) {
23   // access the appropriate hashtable bucket
24   int bucket = compute_hash(word);
25   entry* e = hashtable[bucket];
26
27   // find matching entry
28   while (e != NULL) {
29     if (strcmp(e->key, word) == 0) {
30       // increment count
31       e->value++;
32       return;
33     } else {
34       // traverse linked list
35       e = e->next;
36     }
37   }
38   // If no entry found, create one
39   hashtable[bucket] = new_entry(word, 1, hashtable[bucket]);
40 }
```

Figure 1: Extract from a C program that counts the frequency of words in an input string.

bytes, inserting or removing byte sequences, or extracting random parts of another input in the set of parent inputs and splicing it at a randomly chosen location in the parent.

(4) For each child input, run the test program and collect execution counts for each CFG edge. If the child executes some edge more times than any other input seen so far (i.e., it maximizes the execution count for that edge), then add it to the set of parent inputs.

(5) Repeat from step 2 until a time limit is reached.

We walk through an execution of the PerfFuzz algorithm for the word frequency counting program `wf` shown in Figure 1.

Suppose the seed input is the string `"the quick brown fox jumps over the lazy dog"`. This input does not have any special characteristics that exhibit worst-case complexity. All of the 8 distinct words in this input map to distinct buckets in the hashtable, and none are very long. PerfFuzz first runs the program with this input and collects data about which CFG edges were executed. For example, the function `add_word` is invoked 8 times, whereas the `true` branch of the condition on Line 29 is executed only once to increment the count for the word `"the"`.

In step 2, PerfFuzz picks this input and mutates it several times. Let us walk through a few sample mutations to observe the outcome of each mutation.

(1) The character at position 18 is changed from `o` to `i`, yielding the string `"the quick brown fix jumps over the lazy dog"`. Running the program with this input does not increase the execution count for any CFG edge. Therefore, this input is discarded. This is the most common outcome of mutation.

(2) The character at position 7 (the `i` in `quick`) is replaced with a space, yielding the string `"the qu ck brown fox jumps over the lazy dog"`. This operation increases the number of words, so running `wf` with this input leads to an additional execution of the function `add_word`. As no previous input has executed the CFG edge that invokes this function 10 or more times, the input is saved for subsequent fuzzing.

(3) The character at position 16 (the space between `brown` and `fox`) is replaced with an underscore, yielding the string `"the quick brown_fox jumps over the lazy dog"`. The words `brown_fox` and `dog` have the same hash value of 545, causing a collision-resolving linked-list traversal at line 35. As this branch is executed for the first time, this input is also saved.

Note that the last mutation, (3), actually *reduces* the total number of words, and therefore the total end-to-end execution path length. This is important, and we will return to this point later.

Newly saved inputs will be picked in the future as the *parent* for subsequent mutations, and the process repeats. Inputs that maximize the execution count of at least one CFG edge are *favored*; that is, they are picked for fuzzing with higher probability. A favored input may cease to be favored if newer inputs are found with higher execution counts for the same edge. The number of favored inputs at any time is much smaller than the number of CFG edges in the program due to correlations between execution counts of various edges in the program—the same favored input may maximize the execution counts of correlated CFG edges.

Most mutated inputs will not increase execution counts. However, executing a program with a single input is a very fast operation, even in the presence of lightweight instrumentation for collecting profiling data. So, PerfFuzz can make steady progress in a reasonable amount of time. For example, with our experimental setup, `wf` can be executed more than 6,000 times per second on average. Thus in one hour, PerfFuzz can go through over 20 million inputs.

After a predefined time budget expires, PerfFuzz outputs the current favored program inputs and the execution counts for the CFG edges that they maximize (see Table 1 for an example). For the running example, PerfFuzz outputs strings including

`"tvÇ1PFEj??A4A+v!^?^AE!§^?MPttò8dg80ÿ(8mrÿÿÿÿ",`

a single long word which maximizes the execution count of Line 17 in `compute_hash`, as well as

`"t t t t i nv t X t 1 9 t l t l t t t t t",`

a string containing many short words which exercises repeated executions of the function `add_word()`, and

`"t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t t",`

which contains many words that hash to the same bucket as the word `"t"`, exposing the worst-case complexity due to repeated

traversals of a long linked list. Section 5.1.2 describes in detail the results of running PerfFuzz on `wf-0.41`.

An important feature of PerfFuzz is that it saves mutated inputs if they maximize the execution count for any CFG edge, even if the mutation reduces the total execution path length. This is in contrast to previous tools which use a greedy approach and consider only increases in total path length [49]. This feature helps PerfFuzz find inputs exercising worst-case behavior even when the performance response of the program is non-convex. For example, finding inputs with many hash collisions in the example above usually requires reducing the total path length when discovering the first few collisions—refer to mutation 3. But, the total path length becomes much larger once multiple collisions are found due to the quadratic increase in the number of linked-list node traversals. Empirical results in Section 5.1.2 support the importance of this multi-objective approach.

3 THE PERFFUZZ ALGORITHM

We now describe the PerfFuzz technique formally. Algorithm 1 outlines the high-level input generation strategy. The high-level algorithm is based on the coverage-guided greybox fuzzer AFL [60].

The goal of PerfFuzz is to generate inputs which achieve high *performance values* associated with some *program components*. To generate inputs exhibiting high computational complexity, we take the program components to be CFG edges and the values to be their execution counts. The PerfFuzz algorithm can be easily adapted to maximize a variety of values for different program components: the number of bytes allocated at `malloc` statements, the number of cache misses or page faults at memory load/store instructions, the number of I/O operations across system components, etc.

PerfFuzz is given a program, p, and a set of initial seed inputs (*Seeds*). These seed inputs are used to initialize a set of *parent inputs*, denoted $\mathcal{P}$ (Line 1). Inputs in set $\mathcal{P}$ form the base from which new inputs are generated via mutation.

PerfFuzz then considers each input from the set $\mathcal{P}$ (Line 4) and probabilistically decides whether or not to select that input for mutational fuzzing (Line 5). The selection probability fuzzProb is 1 for an input that is currently *favored* (maximizes a performance value, detailed in Definition 5) and low otherwise.

Each time a parent input is chosen for fuzzing, PerfFuzz determines a number of new child inputs to generate (Line 6). It generates these children by mutating the chosen parent input (Line 7). The number of child inputs produced and the mutation operations used to produce them are implementation-specific heuristics, which we borrow from AFL (see Section 4).

PerfFuzz then executes the program under test with every newly generated child input (Line 8). During the execution, PerfFuzz collects feedback which includes code coverage information (e.g., *which* CFG edges were executed) as well as values associated with the program components of interest (e.g., *how many times* each CFG edge was executed). If an execution results in new code coverage (newCov) or if it maximizes the value for some component (newMax), then the corresponding input is added to the set of parent inputs for future fuzzing (Line 10). Saving inputs which explore new coverage is key to exploring different program behavior when the program component, performance value pairs to be maximized are not simply CFG edges and their hit counts.

Algorithm 1 The PerfFuzz algorithm

Inputs: program p, set of inputs *Seeds*

```
1:  P ← Seeds
2:  t ← 0
3:  repeat                                              ▷ begin a cycle
4:    for input in P do
5:      with probability fuzzProb(input) do
6:        for 1 ≤ i ≤ numChildren(p, input) do
7:          child ← mutate(input)
8:          feedback ← run(p, child)
9:          if newCov(feedback) ∨ newMax(feedback) then
10:           P ← P ∪ {child}
11:         t ← t + 1
12: until given time budget expires
```

Once PerfFuzz completes a full cycle through the set $\mathcal{P}$, it simply repeats this process until a given time budget expires (Line 12).

We now define a series of concepts that are required to precisely describe what it means for an input to maximize a value associated with a program component (i.e., satisfy newMax) and for an input to be *favored*.

Definition 1. A *performance map* is a function $perfmap : \mathcal{K} \to \mathcal{V}$, where $\mathcal{K}$ is a set of keys corresponding to program components and V is a set of ordered values ($\leq$) corresponding to performance values at these components.

Given a $\mathcal{K}$ and $\mathcal{V}$, $perfmap_i$ is the performance map derived from the execution of input i on program p. As outlined in the beginning of this section, the sets $\mathcal{K}$ and $\mathcal{V}$ have deliberately been left abstract to make the algorithm flexible to different program component, performance value pairs.

Definition 2. The *cumulative maximum map* at time step t is a function $cumulmax_t : \mathcal{K} \to \mathcal{V}$. It maps each program component to the maximum performance value observed for that component across all inputs generated up to time t. Precisely, if $\mathcal{I}_t$ is the cumulative set of inputs executed up to time step t, then:

$$\forall k \in \mathcal{K} : cumulmax_t(k) = \max_{i \in \mathcal{I}_t} perfmap_i(k).$$

Now we can get to the fundamental concepts which allow PerfFuzz to achieve its testing goal.

Recall from Line 10 that PerfFuzz adds an input to $\mathcal{P}$ if it achieves a new maximum or if it achieves new coverage. We will discuss the notion of coverage in Section 4, as it is implementation-specific. The first key to the PerfFuzz algorithm is saving inputs which achieve a new maximum compared to previously observed values. In terms of *cumulmax*, an input has a new maximum if:

Definition 3. The function newMax will return true for a newly generated input i at time step t if the following condition holds:

$$\exists k \in \mathcal{K} \text{ s.t. } perfmap_i(k) > cumulmax_t(k).$$

The second key to the PerfFuzz algorithm is the selection of inputs from $\mathcal{P}$ to mutate. To define the selection probability of an input, fuzzProb, we must first define the concept of *favoring*.

Definition 4. An input i *maximizes* a performance value for some component k if and only if its performance profile registers the maximum value observed for that component so far:

$$maximizes_t(i,k) \Leftrightarrow perfmap_i(k) = cumulmax_t(k).$$

Definition 5. An input i is *favored* for fuzzing at time step t if and only if it maximizes a performance value for some component:

$$favored_t(i) \Leftrightarrow \exists k \in \mathcal{K} \text{ s. t. } maximizes_t(i,k)$$

The favoring mechanism is a heuristic that allows PerfFuzz to prioritize fuzzing those inputs that maximize the performance value of some program component. The intuition behind this is that these inputs contain some characteristics that lead to expensive resource usage in some program components. Thus, new inputs derived from them may be more likely to contain the same characteristics. With this, we can define the probability that an input will be selected as a parent for fuzzing:

Definition 6. The *selection probability* of an input i at time t is:

$$\textsc{fuzzProb}_t(i) = \begin{cases} 1 & \text{if } favored_t(i) \\ \alpha & \text{otherwise} \end{cases}.$$

That is, favored inputs are always selected, and α is the probability of selecting a non-favored input. In our experiments we use $\alpha = 0.01$.

4 IMPLEMENTATION

In this section, we fill in some concrete implementation details that were omitted in the general algorithmic description. PerfFuzz is built on top of American Fuzzy Lop (AFL) [60], a state-of-the-art coverage-guided mutational fuzzing engine. As such, many implementation details are inherited from AFL.

Choosing the number of child inputs to produce (Line 6 in Algorithm 1). To determine the number of children to produce from a parent input, PerfFuzz uses the same heuristics as AFL. These heuristics include producing more children for inputs that have wider code coverage or are discovered later in the fuzzing process.

Mutating inputs (Line 7 in Algorithm 1). PerfFuzz has no domain-specific knowledge of input structure: it simply views inputs as sequences of bytes. Thus, the mutation strategies that PerfFuzz uses work at the byte-sequence level. These strategies are the same as those performed by AFL. In particular, in our evaluation, PerfFuzz performs only *havoc* mutations, which work as follows.

Let i be the parent input. Choose a number, m, of mutations to apply (m is chosen randomly from powers of two between 2 and 128) . Then, from $i_0 = i$, produce a series of mutated inputs $i_j = mutate\_once(i_{j-1})$. The final mutated input is $i' = i_m$, which is returned to the main fuzzing loop in Line 7 of Algorithm 1 for program execution. The function *mutate_once* chooses a random 1-step mutation and applies it to the input it is given. The mutations applied by *mutate_once* include, amongst others

- Bitflips/byteflips at random locations.
- Setting bytes to random or interesting (0, `MAX_INT`) values at random locations.
- Deleting/cloning blocks of bytes.

PerfFuzz also retains AFL's input-splicing mutations stage, more commonly called a *crossover* mutation. For a parent input i, a splicing mutation chooses a random input i' in $\mathcal{P}$ and pastes a random sub-sequence from i' at a random offset in i. This stage runs only when PerfFuzz has not recently discovered new coverage or maximizing inputs.

New program coverage. As illustrated in Line 9 of Algorithm 1, PerfFuzz saves inputs that have new maxima (Definition 3) as well as those that achieve new code coverage. In our implementation, PerfFuzz uses the same *new coverage* definition of AFL, which works as follows.

AFL inserts instrumentation into the program that assigns a pseudo-unique ID to every edge in the control-flow graph (CFG) of the program. During program execution, the instrumentation uses an 8-bit counter to keep track of the number of times that each CFG edge was traversed. AFL simplifies the hit counts of each CFG edge into one of 8 buckets: hit 1 time, 2 times, 3 times, 4–7 times, 8–15 times, 16–31 times, 32–127 times, or 128–255 times. Then, an input has new coverage if it either:

- visits a new CFG edge, or
- hits a known CFG edge a new *bucketed* number of times.

This bucketing strategy is an "implementation artifact" [61], which allows AFL to quickly calculate major differences in coverage. As this definition of new coverage has had success in the past, we did not seek to modify it.

Note that with this definition, an input that achieves new coverage may not have a new maximum and vice versa. For example, let e represent a CFG edge. An input hitting e 10 times when e has only been hit 20 times by previously generated inputs achieves new coverage but not a new maximum. On the other hand, an input hitting e 190 times when e has already been hit 130 times achieves a new maximum but not new coverage.

Performance map. In our current implementation, the *performance map* sent back to the program has $\mathcal{K} = \mathcal{E} \cup \{total\}$ and $\mathcal{V} = \mathbb{N}$, where $\mathcal{E}$ is the program's set of CFG edges and *total* is an additional key. For an input i, for each $e \in \mathcal{E}$, $perfmap_i(e)$ is the total number of times the program executes e when run on input i, and $perfmap_i(total) = \sum_{e \in \mathcal{E}} perfmap_i(e)$. The purpose of the *total* key is to save inputs which have high total path length.

To produce this performance map, we simply augmented AFL's *LLVM-mode* instrumentation, which inserts the coverage instrumentation described above into LLVM IR. Our augmented instrumentation still creates the usual coverage map, whose keys are in $\mathcal{E}$ and whose values are their 8-bit hit counts. Additionally, our augmented instrumentation creates the performance map outlined above, with values as 32-bit integers.

5 EVALUATION

In our evaluation of PerfFuzz, we seek to answer the following research questions:

RQ1. How does PerfFuzz compare to single-objective complexity fuzzing techniques such as SlowFuzz [49]?

RQ2. Is PerfFuzz more effective at finding pathological inputs than fuzzing techniques guided only by coverage?

RQ3. Does the multi-dimensional objective of PerfFuzz help find a range of inputs that exercise distinct hot spots?

We chose four real-world C programs as benchmarks for our main evaluation: (1) `libpng-1.6.34`, (2) `libjpeg-turbo-1.5.3`, (3) `zlib-1.2.11`, and (4) `libxml2-2.9.7`. We chose these benchmarks as they are (a) common benchmarks in the coverage-guided fuzzing literature (b) fairly large—from 9k LoC for `zlib` and 30k LoC for `libpng` and `libjpeg`, to 70k LoC for `libxml`—and (c) had readily-available drivers for `libFuzzer`, an LLVM-based fuzzing tool [4]. The availability of good `libFuzzer` drivers was key to being able to fairly compare PerfFuzz to SlowFuzz [49] in Section 5.1. While AFL-based tools need only a program that accepts standard input or an input-file name, `libFuzzer`-based tools rely on a specialized driver that directly takes in a byte array, does not depend on global state, and never exits on any input. Creating drivers with this second characteristic from command-line programs is especially tricky. The particular drivers we chose (from the OSS-fuzz project [7]) exercised the PNG read function, the JPEG decompression function, the ZLIB decompression function, and the XML read-from-memory function.

For each of these benchmarks, we ran PerfFuzz (and the tools with which we compare it) for 6 hours on a maximum file size of 500 bytes. AFL ships with sample seed inputs in formats including PNG, JPEG, GZIP and XML; we simply used the same inputs as seeds for our evaluation. We chose the maximum size of 500 bytes as it was an upper bound on all the seeds that we considered. As the fuzzing algorithm used by PerfFuzz as well as other tools is non-deterministic, we repeated each 6-hour run 20 times to account for variability in the results.

For our evaluation on discovering worst-case algorithmic complexity as a function of varying input sizes (Section 5.1.2), we used three micro-benchmarks: (1) insertion sort (because it was provided as the default example in the SlowFuzz repository), (2) matching an input string to a URL regex [17] using the PCRE library, and (3) `wf-0.41` [6], a simple word-frequency counting tool found in the Feodra Linux repository.

To evaluate PerfFuzz against other techniques, we measure one or both of the *maximum path length* and the *maximum hot spot*, where appropriate. More precisely, if $\mathcal{E}$ is the set of CFG edges in the program under test, and $\mathcal{I}_t$ is the set of inputs generated by a fuzzing tool up to time t, then:

Definition 7. The *maximum path length* is the longest execution path across all inputs generated so far.

$$\text{max. path length} = \max_{i \in \mathcal{I}_t} \sum_{e \in \mathcal{E}} perfmap_i(e).$$

Definition 8. The *maximum hot spot* is the highest execution count observed for any CFG edge across all inputs generated so far.

$$\text{max. hot spot} = \max_{i \in \mathcal{I}_t} \max_{e \in \mathcal{E}} perfmap_i(e).$$

These two values allow us to get a grasp of the overall computational time complexity of generated inputs (the path length) as well as whether it is driven by a particular program component (the hot spot) without having to look at the entire distribution of execution counts of CFG edges, which is not practical to do over time.

5.1 Comparison with SlowFuzz

SlowFuzz [49] is a fuzz testing tool whose main goal is to produce inputs triggering algorithmic complexity vulnerabilities. Like PerfFuzz, SlowFuzz is also an input-format agnostic fuzzing tool for C/C++ programs; therefore, we believe it is the most closely related work to practically compare against.

The objective of SlowFuzz is one-dimensional: to maximize the total execution path length for a program. As such, it serves as an important candidate for evaluating the coverage-guided multi-objective maximization of PerfFuzz against a traditional single-objective technique.

There are two other main algorithmic differences between SlowFuzz and PerfFuzz. First, PerfFuzz produces many (typically at least thousands, often tens of thousands) of inputs from one chosen parent input (Line 6 of Algorithm 1). SlowFuzz instead produces one mutant for each parent. This reduces the importance of selecting inputs to fuzz. Thus, while PerfFuzz prioritizes inputs to fuzz using the concept of *favored* inputs (Line 5 of Algorithm 1), SlowFuzz randomly selects a parent input to fuzz. Second, PerfFuzz applies AFL's *havoc* mutations (as detailed in Section 4) to the input. SlowFuzz learns which mutations were successful in producing slow inputs in the past, and applies these more often.

Finally, SlowFuzz is built on top of on `libFuzzer` [4], an LLVM-based fuzzing tool. In practice, `libFuzzer` is faster than AFL, running more inputs through the program per second; therefore, SlowFuzz usually produces more inputs than PerfFuzz in the same time span. Nonetheless, in our evaluation, we run both PerfFuzz and SlowFuzz for the same amount of time.

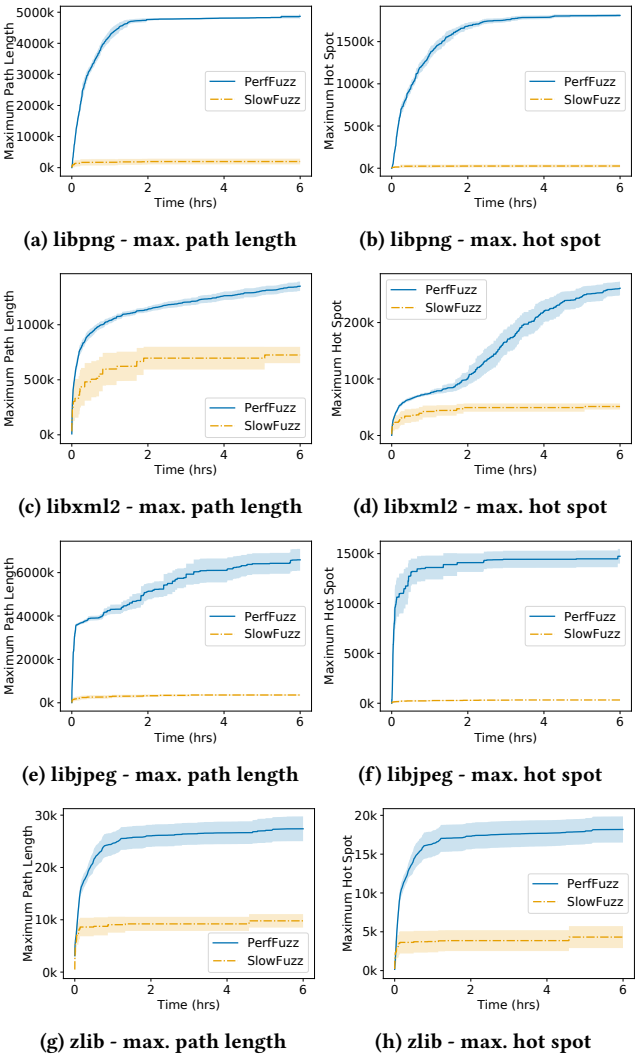
We compare PerfFuzz with SlowFuzz on two fronts. First, we evaluate PerfFuzz and SlowFuzz on their ability to maximize total execution path lengths as well as the maximum hot spot on the four macro-benchmarks described above. Second, we compare the ability of PerfFuzz and SlowFuzz to find inputs that demonstrate worst-case algorithmic complexity in micro-benchmarks which are known to have worst-case quadratic complexity.

In all runs of SlowFuzz, we used the arguments provided in the `example` directory, except that we used the "hybrid" mutation selection strategy. This was the strategy used in SlowFuzz's own evaluation [49], and we found that it performed best on a selection of micro-benchmarks in our initial experiments.

5.1.1 Maximizing Execution Counts. Figure 2 shows the progress made by PerfFuzz and SlowFuzz during 6-hour runs in maximizing total path length (on the left) and the maximum hot spot (on the right). The lines in the plot represent average values over 20 repeated 6-hour runs, while the shaded areas represent 95% confidence intervals, calculated with *Student's t-distribution*.

It is clear from Figure 2 that PerfFuzz consistently finds inputs that are significantly worse-performing than SlowFuzz's by both the evaluated metrics—the maximum path lengths found by PerfFuzz are 1.9×–24.7× higher and the maximum hot spots are 5×–69× higher. This is in spite of the fact that SlowFuzz produces more inputs in each of this 6-hour runs (from 1.7× more for `libxml2` to 17.7× more for `libjpeg-turbo`).

The results show that not only is PerfFuzz better than SlowFuzz at finding hot spots, for which the PerfFuzz algorithm is tailored, but that PerfFuzz is superior to SlowFuzz even for finding inputs



(a) libpng - max. path length (b) libpng - max. hot spot

(c) libxml2 - max. path length (d) libxml2 - max. hot spot

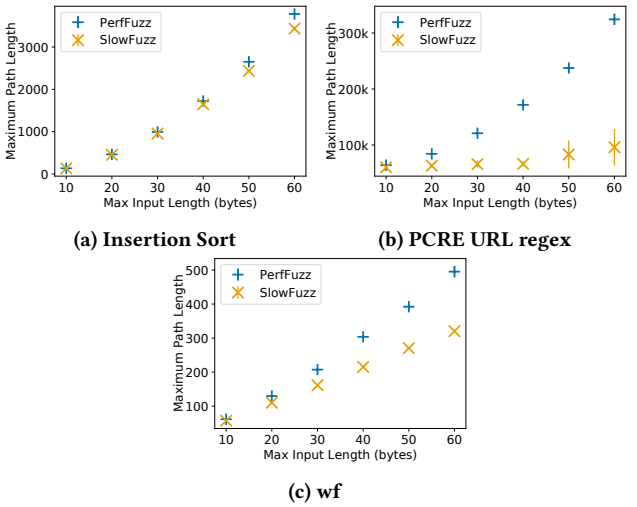
(e) libjpeg - max. path length (f) libjpeg - max. hot spot

(g) zlib - max. path length (h) zlib - max. hot spot

Figure 2: PerfFuzz vs. SlowFuzz on macro-benchmarks: maximum path length and maximum hot spot found throughout the duration of the 6-hour fuzzing runs. Lines and bands show averages and 95% confidence intervals across 20 repetitions; higher is better.

that maximize total path length, for which SlowFuzz is tailored. Intuitively, we believe that this is because the total path length is not a convex function of input characteristics; a greedy approach to maximizing total path length is likely to get stuck in local maxima. In contrast, PerfFuzz saves newly generated inputs even if the total path length is lower than the maximum found so far, as long as there is an increase in the execution count for some CFG edge. Thus, the multi-dimensional objective of PerfFuzz allows it to perform better global maximization of total path lengths.

5.1.2 Algorithmic Complexity Vulnerabilities. SlowFuzz was designed to find algorithmic complexity vulnerabilities, where programs exhibit worst-case behavior that is asymptotically worse than their average-case behavior. Such programs pose a security risk if they process untrusted inputs: an attacker can send carefully



(a) Insertion Sort (b) PCRE URL regex

(c) wf

Figure 3: PerfFuzz vs. SlowFuzz on micro-benchmarks: maximum path length found with given time budget, for varying input sizes; higher is better.

crafted inputs that exercise worst-case complexity and exhaust the victim's computational resources, resulting in a Denial-of-Service (DoS) attack [23]. We now show that PerfFuzz can also address this use case, and in fact can out-perform SlowFuzz in some cases.

We considered three micro-benchmarks: (1) insertion sort on an array of 8-bit integers, which is the only benchmark provided in the SlowFuzz repository, (2) matching an input string against a regular expression to validate URLs using the PCRE library, and (3) `wf-0.41`, the word-frequency counting program from the Fedora Linux repository. These benchmarks are very similar to those used to evaluate SlowFuzz. Each of these micro-benchmarks have an average-case run-time complexity that is linear in the size of the input, and a worst-case complexity that is quadratic.

For each of these benchmarks, we varied the upper bound on the input size between 10 and 60 bytes with 10-byte intervals. We then ran each tool on the micro-benchmarks for a fixed duration: 10 minutes for insertion sort and 60 minutes for PCRE and `wf`. In all cases, we provided a single input seed: a sequence of zero-valued bytes of maximum length for insertion sort and PCRE (these represent trivial base cases), and (truncations of) the string "`the quick brown fox jumps over the lazy dog`" for `wf`, as it leads to average-case performance. For each input length, we performed 20 runs to account for variability. Finally, we measured the maximum path length observed over all the inputs produced in these runs.

Figure 3 shows the results of these runs: points plot the average maximum path length, while lines show 95% confidence intervals.

For insertion sort, for all input lengths, PerfFuzz found a significantly (at 95% confidence) longer maximum path length, but as Figure 3a shows, the difference is minimal for small input lengths. For input lengths 10 and 20, PerfFuzz consistently found the worst-case—a reverse-sorted list—while SlowFuzz had non-zero variance in its results. Figure 3a also shows that for larger input sizes, PerfFuzz finds lists that require more comparisons to sort than SlowFuzz. Overall, both tools discover the worst-case quadratic time complexity for this benchmark.

However, in Figure 3b we see a major difference between the worst-case inputs found by PerfFuzz and SlowFuzz on the PCRE URL benchmark. PerfFuzz finds inputs that lead to worst-case quadratic complexity, while SlowFuzz finds only a slight super-linear curve. An example of an input found by PerfFuzz that had maximum path length in one of the 50-byte runs was:

```
fhftp://ftp://ftp://ftp://f.m.m.m.m.m.m.m.m.m.
```

This is remarkable because the seed input was an empty string and PerfFuzz was not provided any knowledge of the syntax of URLs. On the other hand, SlowFuzz has difficulty in automatically discovering substrings such as `ftp` in the input string. We suspect that this is because of its one-dimensional objective function, which does not allow it to make incremental progress in the regex matching algorithm unless there is an increase in *total* path length. Additionally, Figure 3b shows that there is much more variance in SlowFuzz's performance (see large confidence intervals for length 50 and 60) on this benchmark, indicating that any such progress likely relies on a sequence of improbable random mutations.

`wf` is a much harder benchmark, as the worst-case behavior is only triggered when distinct words in the input string map to the same hash-table bucket (ref. Section 2). Figure 3c shows that PerfFuzz clearly finds inputs closer to worst-case time complexity in the given time budget. We noticed that in nearly all runs (i.e., 19 of the 20 runs for 60-byte inputs), PerfFuzz produced inputs with a very peculiar structure: first a few distinct words with the same hash code, then a single 1-letter word repeated multiple times. For example, PerfFuzz generated this input in one of its runs:

```
t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t
```

What is amazing about this input is how precisely it exercises worst-case complexity. First, a small word is inserted into some hash bucket. Then, the next few words have the exact same hash code and are inserted at the front of the linked list in that bucket; the first word is now the last node in this linked list. Finally, the repeated occurrences of the first word cause `wf` to traverse the entire linked list multiple times. The worst inputs produced by SlowFuzz had some hash collisions, but still had several different hash codes and no traversal-stressing structure like the input above.

Overall, we see that in the same time constraints, PerfFuzz is able to find inputs with significantly longer paths than SlowFuzz, and can out-perform SlowFuzz in discovering inputs exercising near worst-case algorithmic complexity.

5.2 Comparison with Coverage-Guided Fuzzing

With the insight that PerfFuzz's efficacy is in part due to its multi-objective. coverage-guided progress, we ask whether PerfFuzz performs better than just AFL off-the-shelf. To evaluate this aspect, we ran AFL on our four C macro-benchmarks. Like PerfFuzz, AFL was configured to use only havoc mutations (`-d` option), because this configuration has been shown to result in faster program coverage [62]. This experiment tests the value-add of PerfFuzz's performance maps and maximizing-input favoring heuristics.

We begin by looking at the evolution of the maximum hot spot found by each technique through time, shown in Figure 4. For the `libpng`, `libjpeg-turbo`, and `zlib` benchmarks (Figures 4a, 4c, 4d), we see that PerfFuzz rapidly finds a hot spot with a significantly higher execution count. For the `libxml2` benchmark (Figure 4b),

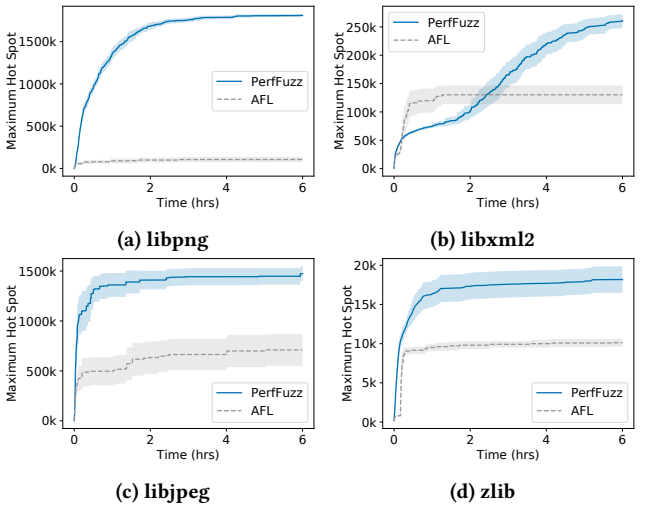


Figure 4: PerfFuzz vs. AFL: Time evolution of the maximum hot spot through the 6-hour runs. Lines and bands show averages and 95% confidence intervals across 20 repetitions. Higher is better.

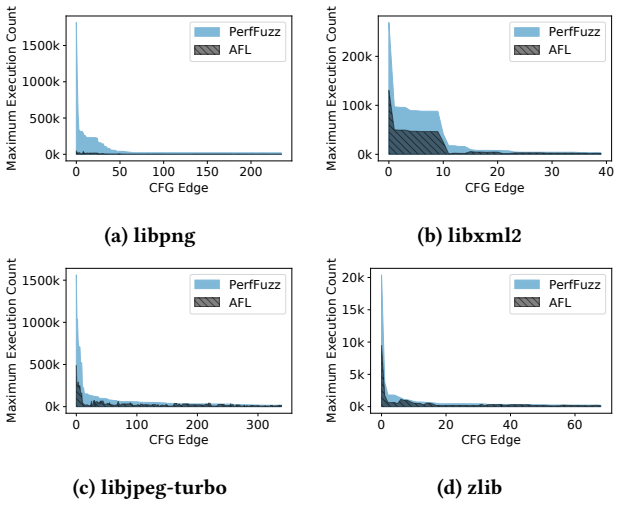


Figure 5: Distribution of maximum execution counts across CFG edges, as found by by PerfFuzz and AFL after 6-hour runs. Plots show median values of this measurement across 20 repetitions.

AFL initially finds a hot spot with higher execution count, but quickly plateaus. On the other hand, PerfFuzz finds a hot spot with over 2× higher execution count after 6 hours. Overall, Figure 4 demonstrates that PerfFuzz's performance-map feedback has a significant effect on its ability to generate pathological inputs, exercising hot spots with 2×–18× higher execution counts.

Figure 4 shows only the execution counts for the maximum hot spot, as this is easy to visualize through time. However, we were curious as to whether the maximum execution counts found by PerfFuzz are significantly higher than those found by AFL over all hot spots in the program. Figure 5 provides this information.

Table 1: A snapshot of the output of PerfFuzz after one 6-hour run on `libpng`. For each of 3 favored inputs, the table shows the top 3 CFG edges—represented by start and end line numbers—by their execution count.

Input #9189		Input #10520		Input #10944	
Exec. count	CFG edge	Exec. count	CFG edge	Exec. count	CFG edge
2,071,824	`pngrutil.c:3715->3715`	289,536	`pngrutil.c:3842->3842`	225,489	`pngread.c:387->396`
274,212	`pngrutil.c:3715->3712`	144,536	`pngrutil.c:3416->3419`	225,489	`pngread.c:405->456`
274,178	`pngrutil.c:3712->3715`	144,536	`pngrutil.c:3419->3404`	225,489	`pngread.c:456->459`

In particular, Figure 5 shows the maximum execution count per CFG edge found by each technique at the end of the 6 hour runs. We plot the median of this measure across the 20 repeated runs. For clarity, we sort the CFG edges by the counts achieved by PerfFuzz and truncate the data to show only those edges with execution counts within 2 orders of magnitude of the maximum hot spot found by PerfFuzz. The omitted tails of the distributions are indistinguishable. Figure 5 confirms that PerfFuzz's gains are not limited to only the maximum hot spot in the program. Across the four benchmarks, there are 453 of the plotted edges which PerfFuzz-generated inputs exercise over 2x more times than AFL-generated inputs, and 238 edges which PerfFuzz-generated inputs exercise over 10x more times.

5.3 Case Studies

PerfFuzz is designed to generate inputs that demonstrate pathological behavior in programs across different program components (in this evaluation, CFG edges). In Section 5.1.2 we saw that the inputs generated by PerfFuzz exercised close-to worst-case algorithmic complexity on micro-benchmarks. We decided to manually analyze the inputs generated by PerfFuzz—in a single run each—on the four macro-benchmarks to see where the hot spots were located and how different input characteristics affected these hot spots.

At the end of each run, PerfFuzz outputs its set of favored inputs—those that maximize the execution count of at least one CFG edge—as well as the execution counts for each CFG edge that it maximizes. Table 1 shows an example of this output: it is a snippet from the results obtained from one run of PerfFuzz on the `libpng` benchmark, showing the top 3 CFG edges by execution count for the top 3 favored inputs.

`libpng`. From Table 1, we can directly look at the source code locations to see which features each input exercises. This alone already highlights different hot spots in the code. For illustration, we look at a snippet from `pngrutil.c` in Figure 6, which shows an excerpt from a function that performs PNG interlacing. The argument `row_info` contains data parsed from the input file. This snippet of code shows two distinct hot spots—sets of input-dependent nested loops—guarded by a `switch` on an input characteristic. Therefore, these hot spots can only be exercised by distinct inputs. As illustrated in Table 1, input #9189 maximizes the number of executions of the inner loop when pixel depth is 1 (Line 3715 of Figure 6), corresponding to a monochrome image. Input #10520, on the other hand, maximizes executions of the inner loop for a pixel depth of 4 (Line 3842 of Figure 6), corresponding to an image segment with 16 color-palette entries. Other inputs stress completely different parts of the code. For example, input #10944 from Table 1 maximizes execution counts for CFG edges in a loop whose bounds are

```
void png_do_read_interlace(png_row_infop row_info, ...) {
   ...
   switch (row_info-> pixel_depth) {
      case 1:
      {
         for (i = 0; i < row_info->width; i++)
3715:        for (j = 0; j < jstop; j++)
                ...
      }
      ...
      case 4:
      {
         for (i = 0; i < row_info->width; i++)
3842:        for (j = 0; j < jstop; j++)
                ...
      }
   }
}
```

Figure 6: Snippet from `pngrutil.c` showing hot spots which can only be exercised by inputs with distinct features.

proportional to the height of the PNG image, as declared in the PNG header: each iteration processes one row of pixels at a time.

From a quick glance at just three favored inputs, we can see that PerfFuzz has enabled us to discover some of the key features which have an effect on the performance of parsing a PNG image independent of the file size, such as the image's geometric dimensions and color depths declared in the header. We repeat this exercise for the other benchmarks, but omit the actual outputs and code snippets from the paper for brevity.

`libjpeg-turbo`. In the `libjpeg` benchmark, we saw a similar distribution of inputs where the hot spots were related to JPEG image properties. For example, one input's hot spot was in processing for an image with 4 : 4 : 0 chroma sub-sampling; the input also had a huge number of columns. Other inputs stressed various points in the arithmetic decoding algorithms. PerfFuzz discovered inputs that stressed processing for both one-pass and multi-pass images.

`zlib`. Compared to image formats, the functionality of the `zlib` decompressor is relatively straightforward. This was reflected by the fact that there were very few edges exercised a huge number of times; that is, there were fewer hot spots. Nonetheless, PerfFuzz discovered an input with a compression factor of nearly 126×, whose processing lead to a long execution path.

`libxml`. The inputs produced by PerfFuzz for the `libxml2` benchmark revealed what appears to be quadratic complexity in the parsing process. The largest hot spot was the traversal of the characters of a string in a string-duplication function. For a 500 byte input, there were 226,512 iterations of this loop. By running the

input, it was quickly apparent that the source of this quadratic complexity came from repeatedly printing out the context of errors in the input. Naturally, inputs generated by random mutation are not well-formed XML files. In fact, these inputs had so many errors that they caused the same work—printing the error context—to be done over and over again. PerfFuzz also stressed error handling code that repeatedly traversed the input backwards to check whether a parent tag had a given name-space; essentially, PerfFuzz learned to produce errors deep in the XML tree, causing pathological behavior.

These case studies indicate that the inputs generated by PerfFuzz lead to non-trivial hot spots being uncovered. The inputs generated for `libxml2` also reveal potential inefficiencies in the program performance. Overall, this analysis suggests that PerfFuzz successfully produces inputs that stress various program functionalities, and may be useful by themselves or as references for creating performance tests on these benchmarks.

6 THREATS TO VALIDITY

Like many other input generation techniques founded in a genetic algorithm-style model, PerfFuzz relies solely on heuristics to produce inputs that achieve its testing goal, which is to exercise pathological program behaviors. In combination with the fact that PerfFuzz is a dynamic technique, this means that PerfFuzz is not guaranteed to find all hot spots in a program or the absolute worst-case behavior for each hot spot it discovers.

In this paper, we focused on discovering bottlenecks due to increase in computational complexity; therefore, we measure execution counts of CFG edges instead of total running time. This helps ensure that our measurements are accurate and deterministic, but also means that the identified bottlenecks may not be the points in which the program spends the most time. This gap could be mitigated by using a different cost model for CFG edges, i.e. to find bottlenecks due to other factors such as I/O operations. While PerfFuzz supports different cost models, we have not performed an empirical evaluation with other types of performance feedback.

Finally, we believe that the reason that PerfFuzz outperforms greedy techniques such as SlowFuzz is due to the ability to overcome local maxima in a non-convex performance space. Although we have anecdotal evidence to back this intuition, such as the observations with the `wf` tool described in Section 2, we have not mapped the performance spaces of our benchmarks to measure their convexity. Doing this would require searching through all possible mutations from each generated input, which is infeasible.

7 RELATED WORK

Apart from SlowFuzz [49], which we have thoroughly evaluated, the tools most similar to PerfFuzz include FOREPOST [33, 41] and GA-Prof [53]. These automatically discover inputs that reveal performance bottlenecks in software, using repeated executions of the test program with candidate inputs. FOREPOST learns rules to select a subset of inputs from a known input space (e.g. a database of records) using unsupervised learning. GA-Prof employs a genetic algorithm, where highly-structured inputs (such as a set of URLs in a transaction) are encoded as genes. In contrast, PerfFuzz requires no domain knowledge since its inputs are represented as byte sequences. PerfFuzz's coverage-guided feedback allows it to automatically discover variety in the input space in order to explore deep program functionality.

Search-based software testing (SBST) [35, 36, 42, 59] leverages optimization techniques such as hill climbing to optimize an objective function. These techniques work well when the objective function is a smooth function of the input characteristics. This is not the case for the general-purpose programs, which are the main targets of PerfFuzz.

Another popular input-generation technique is dynamic symbolic execution (DSE) [8, 11, 18, 21, 29, 30, 40, 52, 55], also known as concolic testing. DSE uses constraint solvers [24] to generate inputs that excercise a given program path. WISE [16] uses DSE to generate inputs that exercise worst-case behavior. This requires an exhaustive search of all program paths to find the longest path up to a bounded input length. Thus, WISE does not scale to large complex programs. PerfPlotter [20] addresses this concern by probabilistically selecting paths to explore, using heuristics to find best-case and worst-case execution paths. Zhang et al. [65] automatically generate load tests using mixed symbolic execution and iterative-deepening beam search. These tools are designed to maximize a single-dimensional objective function (e.g., total path length, total memory consumption). Unlike PerfFuzz, they may not generate a variety of inputs that exercise distinct hot spots in a program.

SpeedGun [50] automatically generates multi-threaded performance regression tests that find bottlenecks due to synchronization. SpeedGun's input space is quite different, as it generates sequences of method calls in a Java class. On the other hand, PerfFuzz does not specifically handle concurrent programs. PerfSyn [56] mutates Java programs to expose bottlenecks in a particular method.

Crosby and Wallach were the first to demonstrate denial-of-service (DoS) attacks that exploit algorithmic complexity vulnerabilities [23]. Subsequent work on detecting and preventing DoS attacks [9, 25, 64] has typically focused on measuring aggregate resource exhaustion and does not specifically identify input characteristics that exploit worst-case algorithmic complexity.

Input-sensitive profiling techniques [22, 31, 63] help estimate the algorithmic complexity of a program function empirically by profiling its execution under varying input sizes. However, such techniques require available inputs. Some previous work has focused on identifying redundant traversals of data-structures [45, 46, 48], a special class of algorithmic complexity bugs. Although these techniques help pin-point program locations that may contain complexity bugs, they do not generate the offending inputs automatically.

Researchers in the real-time and embedded systems community have developed methods to estimate Worst-Case Execution Time (WCET) or to prove that a program's WCET does not exceed specified bounds [10, 14, 26, 34, 57]. However, many of these methods require knowledge of loop bounds in the form of manually provided annotations or programming language restrictions. These methods do not easily apply to arbitrary C programs such as the benchmarks we evaluated in this paper. Further, these methods do not generate the concrete inputs that demonstrate worst-case behavior.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants CCF-1409872 , CCF-1423645 and TWC-1409915, as well as DARPA FA8750-15-2-0104.

REFERENCES

[1] 2011. CVE-2011-3414. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3414

[2] 2011. CVE-2011-4858. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4858

[3] 2014. CVE-2014-5265. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-5265

[4] 2016. libFuzzer. http://llvm.org/docs/LibFuzzer.html. Accessed Jan 2018.

[5] 2017. CVE-2017-9804. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9804

[6] 2017. `wf` - Simple word frequency counter. https://fedora.pkgs.org/27/fedora-x86_64/wf-0.41-16.fc27.x86_64.rpm.html Accessed Jan 2018.

[7] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed Jan 2018.

[8] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven Compositional Symbolic Execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 367–381. http://dl.acm.org/citation.cfm?id=1792734.1792771

[9] João Antunes, Nuno Ferreira Neves, and Paulo Jorge Veríssimo. 2008. Detection and prediction of resource-exhaustion vulnerabilities. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 87–96.

[10] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. 1994. Bounding worst-case instruction cache performance. In *Real-Time Systems Symposium, 1994., Proceedings*. IEEE, 172–181.

[11] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094.

[12] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 46–57. http://dl.acm.org/citation.cfm?id=243846.243857

[13] Thomas Ball, Peter Mataga, and Mooly Sagiv. 1998. Edge Profiling Versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 134–148. https://doi.org/10.1145/268946.268958

[14] Guillem Bernat, Antoine Colin, and Stefan M Petters. 2002. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*. IEEE, 279–288.

[15] Sergey Bratus, Axel Hansen, and Anna Shubina. 2008. *LZfuzz: a fast compression-based fuzzer for poorly documented protocols*. Technical Report. Department of Computer Science, Darmouth College.

[16] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated Test Generation for Worst-Case Complexity. In *Proceedings of the 31st International Conference on Software Engineering*.

[17] Mathias Bynens. 2014. In search of the perfect URL validation regex. https://mathiasbynens.be/demo/url-regex.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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)*.

[19] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*.

[20] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2884781.2884794

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*. 30, 1 (2012), 2.

[22] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-Sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/2254064.2254076

[23] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1251353.1251356

[24] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54 (Sept. 2011), 69–77. Issue 9. https://doi.org/10.1145/1995376.1995394

[25] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. 2015. Radmin: early detection of application-level resource exhaustion and starvation attacks. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 515–537.

[26] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. 2001. Reliable and precise WCET determination for a real-life processor. In *International Workshop on Embedded Software*. Springer, 469–485.

[27] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*.

[28] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*.

[29] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*.

[31] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. 2007. Measuring Empirical Computational Complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 395–404. https://doi.org/10.1145/1287624.1287681

[32] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.

[33] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 156–166. http://dl.acm.org/citation.cfm?id=2337223.2337242

[34] Bogdan Groza and Marius Minea. 2011. Formal modelling and automatic detection of resource exhaustion attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 326–333.

[35] Mark Harman. 2007. The current state and future of search based software engineering. In *2007 Future of Software Engineering*. IEEE Computer Society, 342–357.

[36] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[37] Sam Hocevar. 2007. zzuf. http://caca.zoy.org/wiki/zzuf. Accessed Jan 2018.

[38] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*.

[39] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[40] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *CAV*. 609–615.

[41] Qi Luo, Denys Poshyvanyk, Aswathy Nair, and Mark Grechanik. 2016. FOREPOST: A Tool for Detecting Performance Problems with Feedback-driven Learning Software Testing. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 593–596. https://doi.org/10.1145/2889160.2889164

[42] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, Washington, DC, USA, 153–163. https://doi.org/10.1109/ICSTW.2011.100

[43] Glenford J. Myers. 1979. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

[44] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science* 89, 2 (2003), 44–66.

[45] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 562–571. http://dl.acm.org/citation.cfm?id=2486788.2486862

[46] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 369–378. https://doi.org/10.1145/2737924.2737966

[47] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*.

[48] Rohan Padhye and Koushik Sen. 2017. Travioli: A Dynamic Analysis for Detecting Data-structure Traversals. In *Proceedings of the 39th International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2017.50

[49] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[50] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/2610384.2610393

[51] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[52] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*.

[53] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2015. Automating Performance Bottleneck Detection Using Search-based Application Profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 270–281. https://doi.org/10.1145/2771783.2771816

[54] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 561–578. https://doi.org/10.1145/2660193.2660234

[55] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex - White Box Test Generation for .NET. In *Proceedings of Tests and Proofs*.

[56] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing Programs That Expose Performance Bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 314–326. https://doi.org/10.1145/3168830

[57] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. https://doi.org/10.1145/1347375.1347389

[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[59] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 140–150.

[60] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed Jan 2018.

[61] Michał Zalewski. 2014. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed Jan 2018.

[62] Michał Zalewski. 2016. FidgetyAFL. https://groups.google.com/d/msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ. Accessed Jan 2018.

[63] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 67–76. https://doi.org/10.1145/2254064.2254074

[64] Saman Taghavi Zargar, James Joshi, and David Tipper. 2013. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys & Tutorials* 15, 4 (2013), 2046–2069.

[65] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. 2011. Automatic Generation of Load Tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 43–52. https://doi.org/10.1109/ASE.2011.6100093