Finding Short Slow Inputs Faster with Grammar-Based Search

Ziyad Alsaeed
Qassim University
Department of Information Technology,
College of Computer
Buraydah, Qassim, Saudi Arabia
zalsaeed@qu.edu.sa

Michal Young
University of Oregon
Department of Computer Science
Eugene, Oregon, USA
michal@cs.uoregon.edu

ABSTRACT

Recent research has shown that mutational search with appropriate instrumentation can generate short inputs that demonstrate performance issues. Another thread of fuzzing research has shown that substituting subtrees from a forest of derivation trees is an effective grammar-based fuzzing technique for finding deep semantic bugs. We combine performance fuzzing with grammar-based search by generating length-limited derivation trees in which each subtree is labeled with its length. In addition we use performance instrumentation feedback to guide search. In contrast to fuzzing for security issues, for which fuzzing campaigns of many hours or even weeks can be appropriate, we focus on searches that are short enough (up to an hour with modest computational resources) to be part of a routine incremental test process. We have evaluated combinations of these approaches, with baselines including the best prior performance fuzzer. No single search technique dominates across all examples, but both Monte Carlo tree search and length-limited tree hybridization perform consistently well on example applications in which semantic performance bugs can be found with syntactically correct input. In the course of our evaluation we discovered a hang bug in LunaSVG, which the developers have acknowledged and corrected.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; **Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

performance analysis, mcts, input generation

ACM Reference Format:
Ziyad Alsaeed and Michal Young. 2023. Finding Short Slow Inputs Faster with Grammar-Based Search. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598118

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598118

1 INTRODUCTION

Research in test case generation for performance bugs lags test case generation for other software properties, but recent progress has shown success using mutational fuzzing [22, 26]. For applications driven by richly structured text, grammar-based generation can accelerate the search when implemented atop the same execution and monitoring engine.

Consistent with Lemieux et al. [22], we define *pathological* inputs as inputs that maximize software execution cost subject to a bound on input length. Like prior work, we search for short inputs with maximum cost, expecting that these will show the patterns that cause unacceptable costs if extended to longer inputs. These are reckoned performance "bugs" if the demonstrated performance violates implicit or explicitly stated performance expectations.

Grammar-based fuzzing to search for performance bugs differs in some ways from fuzzing for security bugs. Mutation of arbitrary byte strings is a crucial foundation for security fuzzing, because attacks often use malformed inputs. *Algorithmic* performance bugs are what Aschermann et al. call "deep bugs" [4], beyond the cost of handling parsing errors. We can best search for them in a space richer in well-formed inputs, which may be obtained more efficiently with generation from a model such as a context free grammar.

Building on the work of Lemieux et al., we can make use of performance instrumentation beyond the coverage feedback used by other tools built atop AFL [35]. For some search techniques, it is necessary to transform the enormously variable performance feedback into a known range, which we can do with a streaming quantile estimation. Also, whereas coverage-guided security fuzzing can prune generated inputs to variations with identical coverage feedback, we cannot expect a shorter input to have identical performance feedback. Therefore we limit the length of generated inputs *by construction*.

We have constructed prototypes of two main search techniques for generating short but expensive inputs quickly, with variations, as well as a nearly blind search technique as baselines for comparison. All use the same base technique for generating strings (or derivation trees) from a context-free grammar, and all use instrumentation built on AFL [35] with the extensions developed for PerfFuzz [22]. The first main technique is Monte Carlo Tree Search (MCTS), in which a branch in the search tree corresponds to a selection of a production in the grammar, implemented in the TreeLine prototype. The second technique is genetic search with *splicing* of derivation trees, as in the Nautilus system [4]. This is implemented in the SlackLine prototype, and in a variant SlackLineMC that

weights sampling of the search frontier based on a measure that balances exploration with exploitation, similar to MCTS.

We evaluate our approaches on five real-world applications. The benchmarks include two small applications previously used to compare PerfFuzz to the earlier mutational system SlowFuzz, and three additional systems driven by syntactically rich specification files. We extract a grammar for `graphviz`, a widely-used graph layout system, from its parser source (a yacc grammar). For the scanner generator `flex`, we construct a simple grammar from a documentation page and a restricted grammar that avoids a discouraged construct (dangerous trailing context). The third example, `lunasvg`, is a popular library for rendering vector graphics. During our evaluation of our TreeLine prototype with `lunasvg`, we discovered a hang bug using a grammar obtained from W3C SVG documentation. We reported the bug which the developers acknowledged and repaired [1]. While our grammar-based prototypes perform slightly worse than PerfFuzz on the `wf` micro-benchmark and is on par with the `libxml2` parser (which has a worst case of processing badly malformed input), they perform far better in finding *algorithmic* performance problems in the remaining subject applications.

Our main contributions are:

- Adaptation of grammar-based fuzzing for finding performance issues quickly.
- Adaptations of Monte Carlo Tree Search and for grammar-based inputs generation.
- A simple and practical algorithm for generating or splicing inputs within a length bound.
- Evaluation of our TreeLine, SlackLine, and SlackLineMC prototypes on five real-world examples, comparing them to baseline grammar-based generation techniques and to an implementation of the state-of-the-art performance testing approach PerfFuzz.
- An open-source implementation of our prototypes available in source code and as a Docker container:
  - TreeLine: https://github.com/uo-se-research/treeline.git.
  - SlackLine: https://github.com/uo-se-research/slackline.git.

2 BACKGROUND

Grammar-based fuzzing is a standard approach to increasing the proportion of valid inputs generated by a fuzzer. As a grammar typically describes an infinite set of strings, a grammar-based fuzzer requires a strategy for generating some useful subset of those strings, typically as a form of heuristic search guided by feedback from test case execution.

2.1 Context-Free Grammar

Context-free grammars (CFGs) are a natural and common choice for describing text with rich, recursive structure, including not only programming language source code but also configurations and instructions for many applications. A CFG can be specified as a set of rewrite rules of the form $A \rightarrow \alpha$, where A is a *non-terminal* symbol and α is a *phrase*, a sequence of zero or more non-terminal and terminal symbols. Every non-terminal symbol appears on the left-hand side of at least one rewrite rule. For our purposes, and typically in fuzzing, the terminal symbols represent string literals. We will refer to them as *literals* to avoid confusion with terminal nodes in a search tree. A path in the search tree represents a sequence of *derivation steps* $A \overset{*}{\rightarrow} \alpha$, i.e., a sequence of rewrite rule applications. A terminal node in the search tree will be associated with a phrase α that contains only literals, and will represent the text formed by concatenating those string literals.

Generation of sentences (test case inputs) could be guided in a variety of ways, from some notion of coverage of the grammar to a statistical distribution.

2.2 Monte Carlo Tree Search

If we think of generation as a sequence of choices among rewrite rules, a search technique from sequential turn-based games may be applicable. Monte Carlo Tree Search (MCTS) [5, 7, 8, 11, 20, 25, 28] is a heuristic tree search algorithm that has proven useful in many sequential decision-making domains, especially for games. Most famously, MCTS was the basis for the first competent Go programs [15, 30].

All heuristic search algorithms must choose states along a search frontier for further search. Classic algorithms like A* use a static heuristic evaluation function, e.g., choosing among chess positions by evaluating position strength. They perform poorly when an effective, efficient static heuristic is not available. The key idea of MCTS is to substitute a statistical summary of outcomes from multiple random explorations for a static heuristic. Conceptually it relies on the search tree having "good neighborhoods" and "bad neighborhoods". If some good outcomes are found in a neighborhood, it may be worth exploring further for even better outcomes.

MCTS constructs an explicit search tree, with leaves that represent the frontier of the search. The value of each node for further search is estimated by the UCT (Upper Confidence bound applied to Trees), a weighted average of a an *exploitation* term $\overline{X_i}$, which averages rewards observed below node i in the tree, and an *exploration* term that relates number of times that node has been visited n_i to the number of times its parent has been visited, N_i.

$$\mathrm{UCT}_i = \overline{X_i} + C\sqrt{\frac{\ln N_i}{n_i}}$$

A lower value of the balance proportion C emphasizes more exploration where good outcomes have been observed, while a higher value emphasizes more search in under-explored neighborhoods.

Building the MCTS search tree involves four major steps; *selection*, *expansion*, *simulation (rollout)*, and *backpropagation*.

- *Selection*: Select a child node with maximum *UCT* value. *UCT* values at each node ensure that this can be done by selecting the child with maximal *UCT* value at each step in traversal from the root.
- *Rollout*: If the selected leaf is non-terminal and has been visited fewer than t times, for some fixed threshold t, complete a random sequence of actions leading to a terminal state. The sequence of actions is not recorded, but the ending state is evaluated. In game play, the end state is typically a win or a loss. For test data generation, it is a complete test case input, evaluated by executing an instrumented version of the application under test. Rollout is also called *simulation*.
- *Expansion*: If the selected node has already been visited t times (in which case t rollouts have already been recorded for it), it is promoted to an internal node by producing an edge to a child

node for each possible move from that state. One of its children is arbitrarily chosen for the rollout. The UCT argument n_i for each child will become zero, making the UCT value of these new leaves infinite until they have had at least one rollout.
- *Backpropagation*: Given the reward from a rollout or a known terminal state, backpropagate the reward following the node's ancestors. At each step from a node to its parent, the reward is added to the accumulated known rewards, and the number of visits is increased by 1.

2.3 Derivation Tree Splicing

Instead of thinking of a derivation as a sequence of derivation steps, or a path through a tree of choices, we can consider the whole derivation as a tree with internal nodes representing non-terminal symbols and the children of a node representing the right hand side of a grammar production. With this view, one can splice any subtree at an internal node with another subtree rooted at the same non-terminal. In this way new derivation trees can be generated by hybridizing previously generated derivation trees, as demonstrated by Aschermann et al in the Nautilus system [4].

2.4 Other Related Work

The most closely related work is mutational fuzzing to find performance bugs and grammar-based fuzzing for other purposes, especially for security. SlowFuzz and PerfFuzz are representative of the state of mutational fuzzing for performance testing. PerfFuzz is discussed below and used as a baseline in evaluation. As PerfFuzz dominates SlowFuzz through introduction of multi-objective search, and is otherwise similar, a separate comparison to SlowFuzz is not useful.

Syntactic structure is not the only kind of structure that one might use as a framework to search for pathological inputs. The subfield of combinatorial test case generation searches for combinations of *values* that trigger bugs. GA-Prof uses a genetic algorithm search to find application bottlenecks [29]; Forepost uses unsupervised learning to search primarily for a set of boolean choices [17]. These search spaces are very different from a search of syntactic structure, and likely require different techniques.

Glass-box analysis, typically using symbolic execution, has also been applied to characterizing worst case execution time, which could be viewed as subsuming search for pathological inputs [9, 36]. These are powerful techniques but usually limited to smaller components of an application due to their complexity.

3 APPROACH

Our several variations on grammar-based search for expensive inputs are assembled from a handful of basic building blocks:

- Instrumentation for execution coverage and cost, measured as number of control flow edges (i.e., jumps) traversed in running program. This is built on AFL [35] and the PerfFuzz extensions to AFL [22] for measuring hot spot and total edge count. Except for PerfFuzz itself, which we use as a baseline for comparison with a *non*-grammar-based search, we use only the instrumentation parts of AFL, extended with a network API to receive generated inputs and return measurements.
- Transformation of raw overall cost into a uniform quantile range using a t-digest [13, 14]. We use quantiles rather than raw costs in all combinations that use some form of relative scoring, rather than just tracking the most expensive overall execution or hot spot observed so far.
- A derivation generator that selectively produces only sentences (or their derivation trees) up to a fixed limit on length from a context-free grammar. We pre-process grammars to calculate a minimum length for each production and non-terminal, using a simple and efficient iterative fixed point calculation. Whether we generate input texts directly, or derivation trees, all of our grammar-based sentence generation uses the same pre-processing and the same basic approach to limiting the length of generated inputs.
- The classic upper confidence bound for trees (UCT) formula for balancing exploitation with exploration in search. UCT is used in the classic way in TreeLine, where an internal node is a partial derivation. In SlackLineMC we use it for weighted sampling from a search frontier consistent of complete derivation trees.
- A simpler biasing mechanism that increases or decreases the likelihood of choosing a production depending on the success of its recent uses. This is used both alone as a baseline comparison for the more elaborate search tactics and for "heavy rollouts" in Monte Carlo Tree Search (MCTS).
- Monte Carlo Tree Search, used in TreeLine, with adaptations noted above and some additional adaptations noted in the implementation section.
- Splicing of subtrees selected from a derivation tree forest called a "chunk store", adapted from the approach developed in the Nautilus system [4].

We discuss each of these building blocks in turn, before filling in some additional implementation details in the next section.

3.1 Raw and Scaled Costs

The range of edge execution counts encountered in different applications varies from thousands to billions. This does not pose a particular problem for a classic genetic search that simply retains inputs (or derivation trees) that trigger the highest count seen so far (whether for a hot spot or overall execution). Approaches that attempt to balance *exploitation* of previously observed good results with *exploration* of relatively unexplored portions of the search space, including both MCTS in TreeLine and weighted sampling in SlackLineMC, require rewards in a known range.

We scale the reward for execution cost between 0 and 1 with t-digest, a streaming quantile estimation function [13, 14]. The calculation adjusts dynamically as it receives new observations, so initially an execution cost of 10,000 might be scored as 0.9 (90th percentile), but later in the search the same execution cost of 10,000 might be scored as 0.7 (70th percentile) because the search has discovered many more inputs with costs greater than 10,000 edge hits.

Because the quantile reward for a given execution may vary over time (generally decreasing as the search improves), we must occasionally reassess the value of previously observed inputs. TreeLine periodically discards the entire tree search, but retains the t-digest structure to guide subsequent searches. SlackLine and

SlackLineMC periodically winnow the search frontier (typically referred to as a "buffer" in fuzzing literature), discarding derivation trees and their associated texts if they neither provide new coverage (the criterion used by Nautilus in a similar winnowing step) nor particularly high execution cost as determined by the t-digest.

3.2 Length-Limited Generation

A context-free grammar with recursion can describe an infinite number of possible sentences, with no bound on length. We wish to generate sentences with a predetermined bound on length. Although we think it most appropriate to measure length of grammatically rich input in tokens, we have provisionally adopted measuring length by byte-counting to facilitate fair comparisons with tools (like PerfFuzz) that manipulate raw byte strings, also adopting the same 60 byte limit as in published uses of PerfFuzz.

Whichever measure of length we choose, a literal symbol in the grammar is a string with some fixed cost (always 1 if we are counting tokens). For phrases and non-terminal symbols we can define a *minimum cost* inductively:

- $\textsc{MinCost}(A|B) = \min(\textsc{MinCost}(A), \textsc{MinCost}(B))$
- $\textsc{MinCost}(AB) = \textsc{MinCost}(A) + \textsc{MinCost}(B)$

We precompute minimum costs for each term in the grammar. This can be performed very simply by iterating the calculation starting from impossibly high initial estimates until they reach a fixed point. More efficient fixed point calculation algorithms are known and widely used in data flow analysis and abstract interpretation, but in our experience the simplest iterative algorithm arrives at a precise lower bound for the minimum length of every phrase in a moderately complex grammar in a fraction of a second.

We use these precomputed minimum costs in generation subject to a *budget* for the whole generated sentence. At each step in a derivation, we consider only alternatives that are compatible with the budget. If the minimum cost of the start symbol is no greater than the budget for the generated text, we can ensure that there is always at least one feasible choice at every derivation step.

3.3 TreeLine: Monte Carlo Tree Search

The TreeLine approach is a MCTS in which the root of the search tree is the start symbol of a context-free grammar, and each edge represents a derivation step, similar to other grammar-based test data generators [4, 18, 31, 34]. At leaves of the search tree (the frontier of the search, not terminal nodes representing complete derivations), we use heavy rollouts (as versus uniform random generation) using a simple weighting scheme. Choices in a derivation step are constrained by a *budget* to bound the length of generated inputs, as described above. In place of counting wins and losses as in a typical application of MCTS for game search, we compute quantile scaled rewards based on total execution cost.

In the classic MCTS algorithm for two-player game search, each rollout ends in a *win* or a *loss*, and the value of further exploration below a node in the tree is statistically estimated from the ratio of wins to losses. When searching for expensive inputs, the available signals include the cost of executing the application on a particular input, but also (as shown by Lemieux et al [22]) maximization or novelty in the number of times particular control-flow edges are exercised. These are not 0/1 signals that we can simply count, and there is no a priori way to estimate the range of values we may encounter for a particular application under test. Thus, in place of a ratio of wins to losses, we modulate the feedback within the target application cost range usng quantile ranking with a t-digest.

Parts of TreeLine are a standard application of MCTS. At each step we search for a node with a maximum UCT value by selecting at each internal node the child with the maximum UCT. At a leaf (search frontier) node, we execute a *rollout* (random completion of the derivation); this may be a rollout of zero steps if the node is *terminal*. If a certain number of rollouts (currently 20) have been performed on a leaf that is not terminal, we instead expand the leaf into an internal node by creating a child for each possible derivation step from that node. Feedback from rollouts and from terminal nodes are propagated up the tree in the standard way.

We deviate from the classic MCTS algorithm in not always starting the selection step at the root of the search tree. We keep a buffer of "hot" nodes that have recently yielded a NewCov, NewMax, or NewCost signal, and sometimes restrict the search to descendants of these nodes. This has the effect of exploiting the multiple feedback signals from instrumentation better than any simple arithmetic combination of signals we tried.

A rollout in MCTS can complete a sequence of moves in a completely random fashion, or with some heuristic bias (*heavy rollout*). If we had a very good heuristic for choosing derivation steps, we might prefer a different search algorithm, such as A*; the point of MCTS is to substitute sampling for static evaluation at intermediate steps. Nonetheless even a very weak heuristic can improve the sampling. We keep a simple table of weights for pairs of sequential choices. The weight is significantly increased when that sequence appears in a derivation that produces an input that meets any of the optimization criteria and slightly decreased when it appears in a derivation that does not. Choices in rollouts are proportional to the current weight of that choice in the context of the immediately preceding choice, which tends to focus search around recently fortuitous guesses. Weights are used only in rollouts, not in any other part of the MCTS algorithm.

Scaling rewards with a streaming quantile function keeps them in a predictable range, but rewards given early in the search are rough estimate that systematically overestimate the value of inputs found early, because the search finds more good (expensive) inputs later. Since these rewards have been propagated up the tree into UCT values at each node, they can cause misallocation of search effort. Essentially the search can become stuck in neighborhoods that once looked promising.

We overcome this problem with a surprisingly simple mechanism: Throw the search tree away and start over. The information gained in building an initial tree (and sometimes a second, and a third) is enough to greatly accelerate rebuilding a better tree from the root. The quantile index is retained, so the new scaled rewards reflect all inputs generated from the beginning of the first search. The biases used in rollouts are also retained, so the search tends to make better choices in heavy rollouts.

The only tricky bit is deciding when the search is "stuck" enough to justify restarting. We estimate it by the diversity of recently observed test case execution costs. Again we prefer dynamic tuning to configuration constants that might be over-fit to a particular set

of target applications. We maintain a buffer of recently encountered costs, adjusting its size to the range of costs encountered, and monitor it for diversity when it has filled. A threshold on required diversity grows as a logarithmic function of the number of executions since the current tree search was initiated. When the buffer is full and the diversity (fraction of unique costs encountered) falls below the threshold, the tree is discarded.

3.4 SlackLine: Derivation Tree Splicing

In TreeLine, a derivation is represented as a path in the search tree from root to leaf. An alternative is to represent derivation trees explicitly, so that a subtree of one derivation tree can be substituted for a subtree rooted at the same non-terminal in another derivation tree, as in the Nautilus grammar-based fuzzer [4]. This is the approach taken by SlackLine and SlackLineMC.

SlackLine most closely follows the model of Nautilus, except that it uses our length-limited generation approach. Also, in addition to retaining derivation trees whose associated sentences trigger new coverage, it retains trees that trigger new maximums for hot spots or overall execution edge counts.

All subtrees of a retained derivation tree rooted at a non-terminal are also kept in a "chunk store" for hybridization. These "chunks" are labeled with the length of their associated strings. The *margin* of a generated derivation tree is the overall length budget less the length of the string associated with that tree. The tree may be hybridized by substituting a chunk for one of its subtrees rooted at the same non-terminal, provided the length of the (string associated with) the chunk is not longer than the (string associated with) the subtree to be replaced by more than that margin.

SlackLine uses a standard genetic search algorithm, sweeping through the frontier to generate new derivation trees from previously retained trees. For each tree on the frontier, it randomly selects one node as a "splice point", and attempts to hybridize by substituting a chunk at that point. If no chunk can be substituted (e.g., if all chunks in the chunk store are too long to keep the hybrid tree within the length limit), SlackLine will instead generate a new subtree that does fit within the budget (as there must always be at least one). SlackLine also keeps a simple table of hashes of previously generated inputs, and discards the new tree if it collides with an element in the table (which in our experience is rare, seldom approaching 5% of hybridized trees).

We cannot directly combine tree splicing as used in SlackLine and Nautilus with MCTS that treats derivation steps as moves. We can, however, use the same upper confidence bound for trees (UCT) to guide selection of trees to splice. UCT scores can be propagated in a search tree in which the parent relation is "derived from", even though this pedigree tree is replaced by a simple linear buffer representing the search frontier. SlackLineMC propagates UCT scores in this way and uses weighted sampling to hybridize nodes with higher UCT scores more often. This selection differs from classic MCTS selection starting from the roots (which would be the initial seeds generated to kick off search), but shares the basic idea of balancing exploitation and exploration, crediting derivation trees not only for their own expense but also for bearing expensive children.

4 IMPLEMENTATION

Here we provide a few additional details about our current implementation of TreeLine and SlackLine.

Docker container. As we have benefited from open source repositories for AFL and PerfFuzz, TreeLine and SlackLine are both distributed as an open source repository as well. To further expedite building, testing, and extending then, we have implemented each tool in a Docker container with all the legacy dependencies of AFL.

Two processes. We have implemented the main logic of TreeLine and SlackLine as a Python program communicating over a Unix socket with a version of AFL that builds upon the extensions earlier built by the developers of PerfFuzz. This architecture has a performance cost, not only because the Python interpreter is much slower than compiled C code in AFL, and because AFL is idle between requests, but also because each test execution requires inter-process communication to send a test input to the AFL side and another to receive feedback. This could be ameliorated by building a single process executable with Python's embedding support, but we have not done so yet. Using Python was a decision taken to expedite experimentation and ease of change compared to C. This separation of search logic from the C code base may also be helpful to others who wish to read, extend, or replace our approach in further research.

Timeouts. Although we search for slow test cases, sometimes TreeLine and SlackLine find test input that is *too* expensive. We set a timeout of ten seconds on test case execution. Executions that time out are logged.

Lazy search. We do not exhaustively search for the reachable node with maximum UCT value each time we draw from the buffer of hot nodes. Since UCT values change slowly, on most iterations we look only at a smaller buffer of nodes with the top 10 UCT values. Periodically (every 500 iterations) this buffer is updated.

5 EVALUATION

We experimentally evaluate our implementations of grammar-based search for inputs that trigger slow execution with respect to the following research questions.

RQ1. How does the performance of the grammar-based search approaches compare to mutational fuzzing, especially with limited expenditure of computational effort, on real-world benchmarks?

RQ2. When and how much do the search strategies implemented in TreeLine, SlackLine, and SlackLineMC improve over simpler grammar-based search.

RQ3. Are short (1 hour) grammar-based fuzzing campaigns effective in finding performance issues?

We conduct our experiments on a Docker container hosted on a modest workstation.[1] All prototypes, including PerfFuzz, are built on AFL [35] with the additional instrumentation constructed by the PerfFuzz developers. AFL is configured with default settings for C/C++ applications. Only PerfFuzz itself uses other facilities of

[1] All experiment were executed in a Docker container on a dedicated Dell i7 desktop computer running Ubuntu Linux. The Docker container was given full access to the available memory and CPU cores from the host machine (16GB of RAM and 16 CPU cores). Only one experiment in one Docker container ran at a time. Aside from stopping other user applications for consistency, the workstation was running in its normal state (e.g., it was not isolated from the department network).

AFL (mutating strings, managing the queue of generated inputs, etc) and is integrated into the AFL binary. All other implementations run in a separate Python process, using only the instrumentation facilities of AFL through a simple client-server interface that accepts an input string and replies with performance counts.

We used five real-world C/C++ applications for evaluation. We selected two that were previously used to evaluate PerfFuzz [22]: `wf-0.41` a simple word frequency counter [12], and `libxml2-2.9.7`, the XML parser and toolkit of Gnome [33]. The other benchmarks used by PerfFuzz (e.g., `libpng` and `libjpeg`) expect a binary based input for which our grammar-based tools are inappropriate.

`wf` requires no particular input structure. We construct a trivial grammar of character sequences composed from the English alphabet and whitespace.

While `libxml2` processes structured text, it is only a parser whose behavior is not driven by that structure beyond accepting or rejecting it. In fact fully valid XML is a cheap input for `libxml2`; its taxing inputs are invalid files. To produce both valid and invalid inputs for for `libxml2`, we constructed a simple grammar that includes XML special characters (e.g., opening `<` or closing `>` tags.) and the English alphabet for tags or content.

We add three new benchmarks that are widely used software tools with available source code, and which exemplify the class of application, driven by richly structured input, for which we hope grammar-based search will prove advantageous.

Graphviz dot. `graphviz-2.47.0` [27] is a widely used graph visualization toolkit, comprising several tools for manipulating and visualizing directed graphs. The most widely used of the graphviz tools is *dot*, which finds a layout for a directed graph. Dot is also the name of the graph description language used by the dot application. Conveniently for us, source code for the graphviz toolkit includes a Bison EBNF grammar for the dot graph language. It is a simple exercise to strip C code and use the provided grammar to generate dot descriptions rather than parse them.

The *dot* notation is minimal and forgiving. It is not difficult for AFL-based mutators to find valid *dot* inputs, but we expect grammar-based tools to produce fewer malformed *dot* inputs.

Flex. `flex-2.6.4` [24], a reimplementation and extension of the original Unix tool lex, is a fast and very widely used generator of lexical analyzers. Although the `flex` source code is also provided with a Bison parse specification, much of the parsing logic is encoded in C code rather than the grammar. We found it easy, however, to construct a simple grammar for `flex` specifications from a page of documentation containing examples of each construct.

The syntax of `flex` regular expression patterns is sufficiently complex that we expect grammar-based search to be at an advantage. Moreover, the processing that `flex` performs *after* parsing the input is likely to be more expensive than the parsing, making `flex` a promising candidate for finding "deep" performance bugs.

Lunasvg. Scalable vector graphics (SVG) is a standard XML-based notation for specifying graphics. Most web browsers and many illustration applications, including the Adobe suite, support SVG. `lunasvg-2.3.2` [32] is a popular standalone library for SVG rendering. Although intended primarily for use as a library in other applications, `lunasvg` is distributed with a simple command-line program to render an SVG file.

We composed an SVG grammar from the W3C [10] documentation with a focus on the path element. We reduced the whitespace alternatives as we are exploring semantically relevant choices, not parsing.

In contrast to `libxml2`, malformed input rejected by the parser is not the most taxing input for `lunasvg`. Rather, the computationally difficult tasks include positioning, scaling, and rendering graphics and text in complex relationships. Our expectation is therefore that grammar-based search should be more appropriate than conventional byte-string mutation for finding inputs that slow `lunasvg`, although tools like AFL have proved capable of producing well-formed XML and other structured input.

Subject program sizes. We used the utility `cloc` to count source line of code (SLOC) for the core C and C++ files in each subject program, skipping appropriately marked test files. We count 394 SLOC for `wf`, 13K for `lunasvg`, 22K for `flex`, 191K for `libxml2`, and 1.0M for `graphviz`.

Test conditions: Seed inputs. Prior published evaluations of PerfFuzz used trivial seeds consisting of strings of zero bytes. Such seeds would disadvantage PerfFuzz in a comparison with our tools, so we instead collect seed inputs from official documentation as available. As required by AFL, we adjust the size of seed inputs to be within maximum allowed input budget, either breaking them into multiple valid seeds or removing redundant keywords. In the case of `flex` we omit C code from inputs to keep input reasonably sized and additionally provided PerfFuzz with random but valid seeds produced by TreeLine. In addition, we add seed inputs to each benchmarks for all the grammatical keywords in the grammars used by TreeLine and SlackLine. In the case of `lunasvg`, for example, we added more than 570 seed inputs either generated by our tools or composed manually to fill missing keywords and diversify the sample seeds.

Test conditions: Time budget. Experiments can be conducted based on different computational budgets. For example, we can run TreeLine and SlackLine for T hours or specify the number of target-application execution allowed. We primarily used a time budget of 60 minutes, but also conducted some experiments with longer duration or a limit on the number of executions to evaluate (by comparison with time-bounded experiments) the impact of input generation speed. The 60 minute experiments were repeated 20 times to obtain tight confidence bounds on variation (illustrated by shaded regions in plots). To gain confidence that inputs discovered in a 60 minute experiment are "good enough" in the sense that they reveal patterns similar to those that can be found in a longer fuzzing campaign, we also ran some 360 minute (6 hour) experiments and manually inspected the most expensive inputs discovered by short and long campaigns.

Test conditions: Length budget. The efficiency of grammar-based and mutational fuzzing are impacted by the length bound on generated inputs. Longer inputs take longer to generate, but may be necessary for applications that take richly structured input. A length bound as little as 10 bytes could be enough to find performance issues in `wf`, but for applications like `graphviz` with keywords like

"digraph" and recursively nested structures, longer inputs may be required. We use a budget of 60 bytes as a base across all benchmarks, to be consistent with the setting in published evaluations of PerfFuzz.

Test conditions: Figures of merit. The primary variables we use in evaluation are elapsed time and the maximum observed count of control-flow edges traversed, which we take as a proxy for performance. We collected additional metrics, including "hot spot" (maximum executions of any single control flow edge), introduced by PerfFuzz, and we use the full set of metrics collected by PerfFuzz in the search. As we found effectiveness in finding hot spots exactly paralleled effectiveness in finding overall worst-case performance, all graphs and reported figures are for maximum overall control-flow edge hits.

5.1 RQ1: Grammar-Based Search vs. Mutational Text Fuzzing

Ideally we would compare an implementation of our grammar-based techniques to a variant implementation that is identical in every regard except that it uses mutational text fuzzing and a genetic search (the AFL approach) rather than MCTS or tree splicing with grammar-based generation. PerfFuzz is not quite that ideal comparison, but it is the state-of-the-art tool for finding pathological inputs with a bound budget, and it is built atop the same AFL engine for execution monitoring, with some extensions that we have also used. The potentially confounding implementation differences mostly favor PerfFuzz, as it is implemented in C within the same executable process as its AFL base, whereas our prototypes are implemented in Python and communicate to the AFL-based server through inter-process communication.

We evaluate the benchmarks across the three variants of our tools (TreeLine, SlackLine, and SlackLineMC), as well as PerfFuzz and BiasOnly as a representative of simpler grammar-based search. (We also implemented a completely blind grammar-based generator, but its performance was so poor that we soon adopted BiasOnly as a better representative of simple grammar-based generation.) All of the grammar-based variants use the same grammar.

All benchmarks were run 20 times for one hour. While security fuzzing campaigns often last much longer, we consider an hour a reasonable allocation for a test activity that might be incorporated into a routine testing process, repeated nightly or several times a day. Plots in Figure 1 show averages with 95% confidence intervalues across 20 repetitions. As efficient routine testing might reasonably cut off search before its maximum time allocation has expired, we also gauge for each benchmark when each approach first levels off, with less than 5% progress in a window of 1000 seconds.

Unstructured input: WF. Figure 1 highlights the main outcomes for each benchmark using PerfFuzz, SlackLine, SlackLineMC, and TreeLine. As expected, PerfFuzz does at least as well as grammar-based search for the the unstructured inputs of `wf` and for finding malformed inputs that tax `libxml2`.

A sample expensive input for `wf` generated by PerfFuzz is shown below (The symbols ↩ and ↪ indicate we broke the line to fit the input within space).

```
x\n ï\n l\nM\nw\n u\n f\n b\n â\n Ï\nD\n i\nT\n a\n t\n d\nO\nSS\n v\n e\n y↩
    ↪\n s\nG\n ó\n ñ\n j\n q\n r\n n\n z
```

The main pattern captured hints that maximizing the number of words by placing the shortest possible words into a dedicated line for each leads to a more expensive input. Following the same sampling from SlackLine, SlackLineMC, and TreeLine we obtain the inputs below respectively.

```
t\n u t p\n F\nR\n y\nT\nO\nW\nH\n n\n S4p\n r\n t\nU\n f\n t\nP\n u\nX\nE↩
    ↪\n t\nK\n \n j\n \n\nHn\n l\n t r
```

```
a\n n\n b\n y\nE\n v\n i\nK\n l\nG\nM\n I\n l\n c8Z\nXl\nR\n o\n yd\n p\n d↩
    ↪\n y8 t\nX\nY\n vR\nF\n r\n
```

```
r0\n t\nW9y3u5i8S\nH\nP\n b5\n n\n o6O5\n g\n j\n s1F\n e2c\n x\nU\nG7R↩
    ↪\n I\n p\n 9T\n 5q8Z
```

The input from grammar-based tools has the same general pattern of the high cost input found by PerfFuzz, but PerfFuzz refines it more effectively.

Note that while the grammar-based tools use the full English alphabet, PerfFuzz generates a much wider family of characters by random manipulation of bytes. The PerfFuzz developers report effectiveness at finding hash collisions [22], which the more limited alphabet used in our grammar is less likely to generate. The average maximum cost of input found by PerfFuzz in a one hour run is 1.008x as expensive as the combined average maximum cost found by SlackLine, SlackLineMC, and TreeLine. For this class of application, mutational fuzzing is at least as effective as TreeLine, although the advantage is small.

Malformed input: libxml. While some of the grammar-based searches (but not BiasOnly) are initially more effective finding expensive malformed inputs for `libxml2`, by the end of a 60 minute run PerfFuzz has surpassed TreeLine and is within the confidence bounds of SlackLineMC. Unsurprisingly, a grammar that permits free combination of fragments is not a particularly effective approach for generating malformed inputs.

On the other hand, applications that require structured input (Figures 1c, 1d, 1e, & 1f) to test the core functionality impose a challenge for PerfFuzz. For `lunasvg`, `graphviz` and `flex` the average maximum cost found by SlackLine, SlackLineMC, and TreeLine combined are respectively 2.17x, 11.04x and 2.81x as expensive as the average maximum cost found by PerfFuzz.

Performance fuzzing LunaSVG. We selected LunaSVG because, in addition to being a popular open source library, it consumes highly structured text (XML) and is directed in complex ways by the structure of that text. We anticipated that we might find short SVG specifications that were particularly challenging to render. We did not anticipate that we would find an input that triggered an infinite loop.

When one of the tools hits the timeout limit (10 sec), execution is interrupted. In the case of `lunasvg` all the grammar-based tools eventually reach the timeout limit at approximately 4B edge hits (Figure 1d). We examined a sample of the inputs generated by TreeLine alone (shown below) and found that the inputs generated actually triggers an infinite loop in one of the libraries used by `lunasvg`. We reported the bug. It was accepted by the `lunasvg` developers, who fixed it [1].

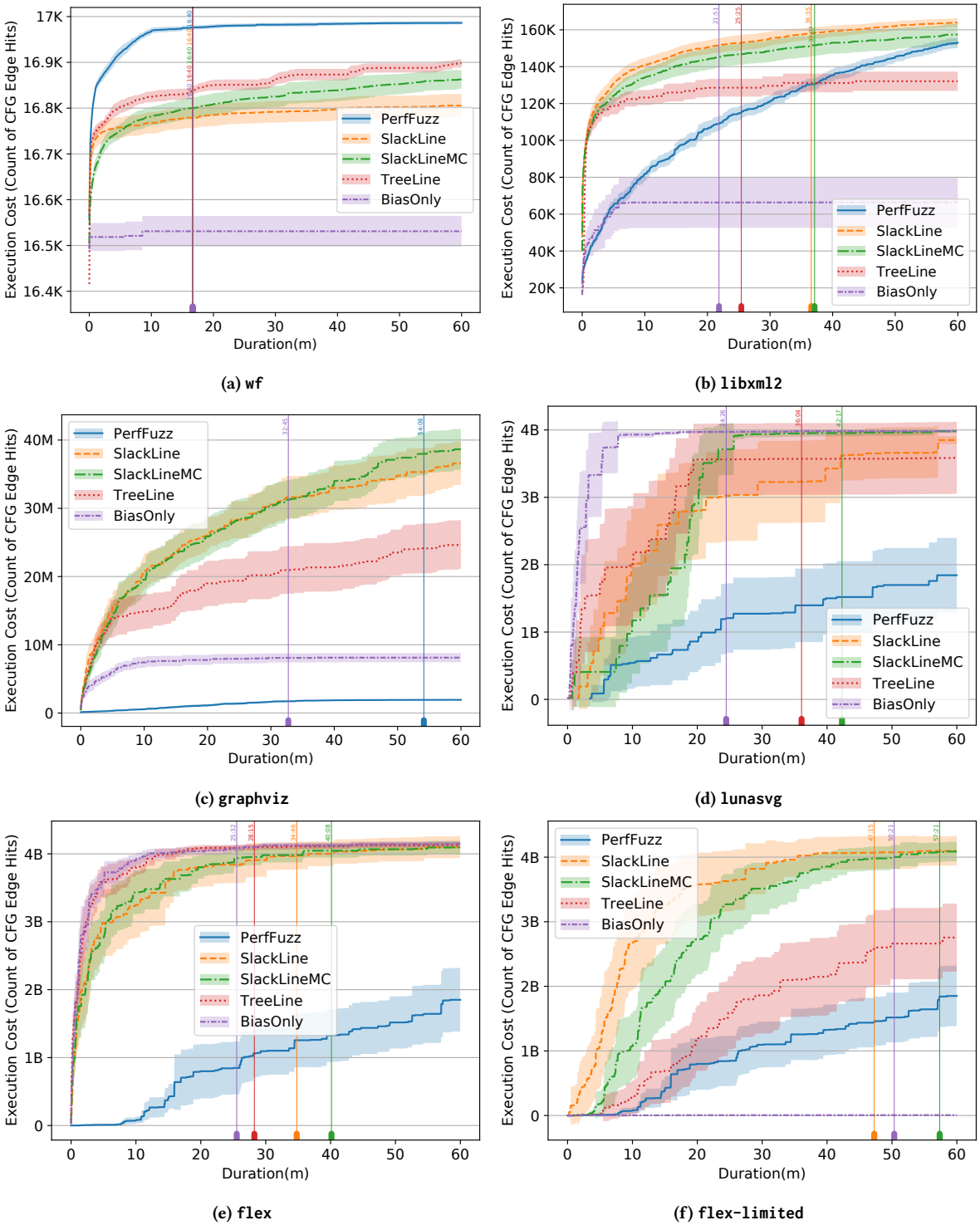
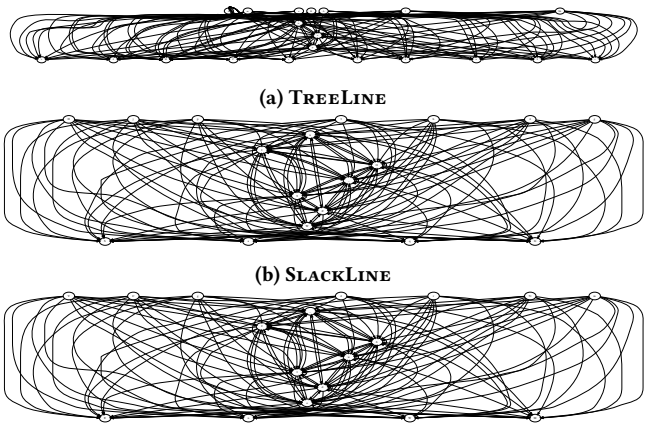


Figure 1: Maximum execution path length (count of control flow edges executed) by elapsed search time for PerfFuzz, SlackLine, SlackLineMC, TreeLine, BiasOnly, within a one hour run. Lines and bands show averages and 95% confidence intervals across 20 repetitions. Vertical lines mark when a moving average of progress in a window of 1000 seconds falls below 5% for the first time. I.e., first stationary area. 10 second timeouts were encountered with `flex` and `lunasvg`. `flex-limited` is the `flex` benchmark re-run with a grammar that does not permit dangerous trailing context (but still encounters timeouts).



(a) TreeLine

(b) SlackLine

(c) SlackLineMC

Figure 2: A graphviz-based rendering of the most expensive input found by TreeLine, SlackLine, and SlackLineMC.

```
<svg><path d="M    9 ,29S206,-2959,+335206,-22-20 2"/></svg>
```

We initially found the hang in `lunasvg` with TreeLine, but subsequently confirmed that all of the grammar-based tools could find it.

Performance fuzzing flex. We encountered timeouts with flex as well (Figure 1e). All the grammar-based tools find inputs that hit the timeout limit in less than 2 minutes across all 20 experiments. Examining a selected random maximum input from TreeLine, we found it triggers the warning for the known "`dangerous trailing context`" [23] from `flex`'s documentation.

```
%%\n.{31}{13}{0,65}/"\]"(\]+) ;\n%%\n
```

Technically the timeouts in flex are not "known bugs", because flex documentation warns that dangerous trailing context can cause the generated scanner to be slow, not that flex itself will be slow. Nonetheless as flex does warn of consequences for excessive use of trailing context, we considered it an uninteresting issue. An advantage of a grammar-based input generator is the ability to steer the grammar off generating such known inputs. If the performance bug is known by the developer and accepted as it is, then a grammar can be tailored to focus on other or more specific modules. Hence, we updated the used `flex` grammar to restrict the reach of the known issue. Figure 1f illustrate the performance of the same grammar-based methods on a limited grammar while PerfFuzz still use all the available seed inputs.

The change in grammar to isolate the known issue clearly effects TreeLine. However, (SlackLine, SlackLineMC, and TreeLine) still find more expensive inputs than PerfFuzz despite the restriction. In fact, SlackLine and SlackLineMC still reach the timeout limit *without* using dangerous trailing context by introducing enormous state machines.

Fuzzing graphviz: Complex graph layouts. No tool generated inputs that caused `graphviz` to exceed the 10 second execution limit for a 60-byte input, but the evaluation does demonstrate finding complex and expensive patterns in structured input (Figure 1c). All the grammar-based tools, even BiasOnly, outperform PerfFuzz.

Benchmark	exec/second			
	SlackLine	SlackLineMC	TreeLine	PerfFuzz
`wf`	201.33	68.85	101.27	2,936.80
`libxml2`	193.96	70.69	94.73	1,507.81
`lunasvg`	8.60	5.20	18.95	255.25
`graphviz`	36.33	12.18	59.81	729.45
`flex`	1.54	0.95	10.10	162.78

Table 1: Average number of target application executions per second (exec/s) for each benchmark across 20 runs for each. Mutational fuzzing in PerfFuzz is much faster.

A representative maximum input from TreeLine (shown below) created a complex cyclic digraph that embeds a complete bipartite digraph on a subset of its nodes (Figure 2).

```
digraph{d:n,e,w,K,A,f,m,Nk,a,p->4,w,a,k,8,H,p,H,r,z,o,i,P,5}
```

To maximize the input cost, it is not enough to create a bipartite digraph; a balance between the bipartite part and cycles must be found. For example, even though the hand crafted inputs below represent a bipartite digraph, a bipartite graph and a graph in which all nodes participate in cycles respectively, are 7.87x to 21.78x cheaper than the input found by TreeLine.

```
digraph{a:n,b,c,d,e,f,g,h,i,j->k,l,m,n,p,q,r,s,t,u,v,w,x,y}
```

```
digraph{o:n,o,o,o,o,o,o,o,o,o->o,o,o,o,o,o,o,o,o,o,o,o,o}
```

SlackLine and SlackLineMC find the same pattern found by TreeLine as shown below respectively. However, they were able to balance the number of outgoing directed edges more effectively to reach even a more expensive case.

```
digraph{h,n,4,g,1,O,_,9,Z,3,W,j,D,p->a,n,W,9,Z,_,O,q,1,G,S;}
```

```
digraph{r,51,_,2,6,w,m,4,1,C,V,d,o,g->_,n,W,M,1,E,o,w,6,2,d}
```

Execution speed differences. A clear advantage of mutational fuzzing of byte streams over grammar-based generation is the speed with which inputs can be generated. AFL is particularly well-engineered to generate and test a vast number of mutants very quickly. The difference in speed of execution between all grammar-based search tools and PerfFuzz ranges from 7.77x times slower for `libxml2` using TreeLine to 171.64x slower for `flex` using SlackLineMC. Some but probably not all of this difference may be due to implementation details (pure C code in PerfFuzz versus a separate Python process for our grammar-based prototype tools). Clearly the grammar-based search tools must generate *much* better inputs on average to compete. The details of execution speed per second is shown in Table 1.

5.2 RQ2: Does Search Strategy Matter?

Our second research question is how much of the performance of our more complex search prototypes, TreeLine, SlackLine, and SlackLineMC, is attributable to their search strategies, as versus being inherent in *any* grammar-based search for performance bugs,

including very simple strategies. If very simple grammar-based generation methods perform as well as SlackLine, SlackLineMC, or TreeLine, then the more complex search strategies are not worthwhile.

All of our grammar-based test generation prototypes use the same core algorithm for producing only strings (or their derivation trees) within a length bound. We briefly experimented with a prototype that naively made random grammar decisions. It was predictably terrible, and we soon abandoned it. In its place as a baseline for comparison to the more complex search strategies we constructed a prototype that makes choices of productions weighted by the recent success of sentences generated from those productions. Each time a generated input garners positive feedback from instrumentation, the weights of all the productions in the derivation of that input are increased substantially. Each time a generated input does not garner any positive feedback (neither new coverage, nor a new hot spot max, nor a new overall max), the weights of all the productions in the derivation of that input are decreased slightly. This is the same biasing mechanism used in heavy rollouts in TreeLine.

Sometimes the simple BiasOnly approach works quite well, and even in some cases better than the more sophisticated search techniques. However, it is extremely inconsistent. BiasOnly finds the same timeout that the other grammar-based techniques find for `lunasvg`, and even faster, but performs poorly on `graphviz`. Experiments with `flex` are particularly telling (Figures 1e and 1f). Although BiasOnly was comparable to SlackLine, SlackLineMC, and TreeLine on the full grammar, when the grammar was restricted to disallow dangerous trailing context (`flex-limited`) the performance of BiasOnly was almost flat and near zero.

We cannot say that the more complex strategies in SlackLine, SlackLineMC, and TreeLine are always better than a simple weighting strategy. To the extent these benchmark examples are representative, we can say that they are more consistent.

5.3 RQ3: Are Short Fuzzing Campaigns Sufficient?

Fuzzing is typically applied in lengthy campaigns to find security vulnerabilities. If performance fuzzing is to be used routinely as an incremental testing technique in software development, it must be more efficient. It need not be the case that a one hour fuzzing campaign finds precisely the same costly inputs as a longer campaign, but the shorter campaigns should expose general patterns that indicate algorithmic performance issues[2].

We ran a smaller number of six hour experiments (matching the time limit in published evaluation of PerfFuzz) to gain confidence that longer runs refined the patterns found in shorter runs, but did not uncover fundamentally different performance issues. Repeating each 6 hour run 20 times would have required months of calendar time. We ran each five times, insufficient for drawing tight confidence bounds around variability, but sufficient to mitigate worries that longer campaigns might uncover fundamentally different patterns.

[2]This section is in part a response to issues raised by ISSTA reviewers. These results were not reviewed with the ISSTA submission, although they were summarized in an author response to reviews.

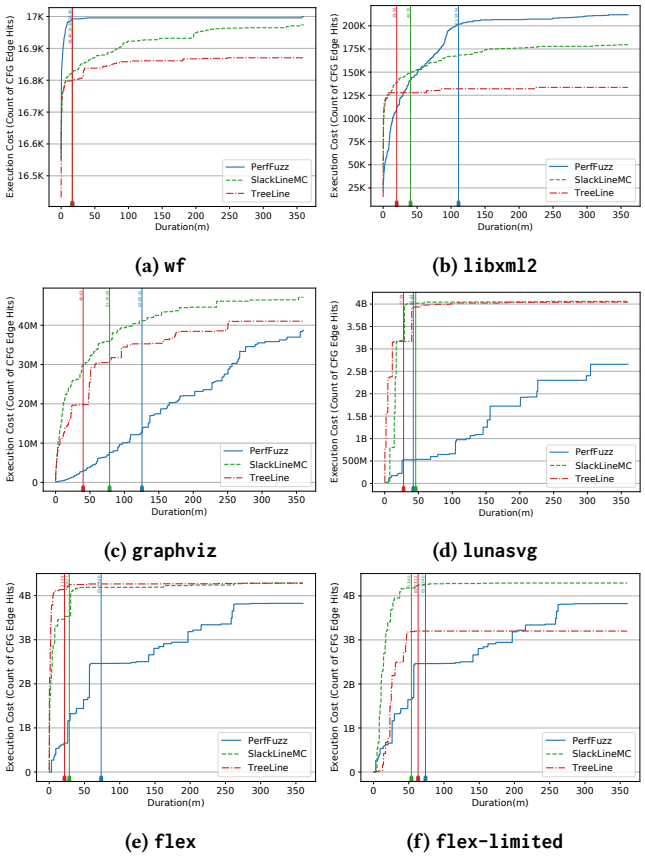


Figure 3: A comparison between PerfFuzz, SlackLineMC, and TreeLine showing maximum path length found throughout the duration of the 6-hours (scaled by minutes). Lines show the averages across 5 repetitions. We do not calculate the confidence intervals in this case due to the small sample size.

The average cost shown in Figure 3 is consistent with the 1-hour run findings demonstrated before. The case of `libxml2` 3b is the only firm case where PerfFuzz finds significantly more expensive input if it is allowed more time. The more expensive inputs found in six hours do not necessarily indicate a new hot spot. For PerfFuzz across the 20 repetitions in the one-hour case, the input below was the most expensive, costing 164, 615 edge-hits.

```
<b><r<o<Q<<s<V<Q<C<s{<Ań<c<V<v<v<a<o<V<h<o<n<o<V<t<w<r<W<ou<
```

Following the same input sampling in the case of six hour runs, PerfFuzz generates a refined input of the same earlier case that costs 214, 105, as shown below.

```
<:><0x0e><:<:<:<:<:<:<:<:<:<:<Ăf<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:↩
↪<:<
```

This is also true in the case of SlackLineMC where the one-hour and six-hours cost 176, 379 and 181, 349, respectively. The two inputs are demonstrated below starting with the one-hour case.

```
<:><_?<:<:<:<:<<:<:<:<<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<\n
```

```
<:><:<<:<:<<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<:<c<:<:<:<:<\n
```

In the case of `graphviz`, for PerfFuzz the most expensive input in the one hour set of experiments cost 1.97M. On the other hand, the most expensive input it finds from the 5 repetitions of six-hours runs cost 44.3M. The two input inputs are shown below respectively.

```
digraph{d {{da\ndaÃf J->TA ->_}\n}->{d -> a:h->7-> X2; } }
```

```
digraph{u{{q Ãž g 7qR.1Y Ã£}4Åđ-0Ã£ÅÃ1 3T}->{4Y 7q 9L 8D;J-7Ãf⟷
    ⟷}Ãñ}
```

This significant variation is absent for TreeLine and SlackLineMC. Taking SlackLineMC as a sample, the most expensive input found in one-hour runs across 20 repetitions cost 50.8M. Following the same sampling procedure for six-hours runs, the most expensive input cost was 53.8M. The two samples are shown below respectively.

```
digraph{r,51,_,2,6,w,m,4,1,C,V,d,o,g->_,n,W,M,1,E,o,w,6,2,d}\n
```

```
digraph{{i,H,r,3,9->F,O,b,0,_->v,u,h,3;}->p,O,q,v,J,t,u,T,P}\n
```

PerfFuzz appears to benefit significantly from six hours runs. TreeLine and SlackLineMC do not; one hour runs (and often much shorter runs) appear to find costly patterns that are only somewhat refined in longer runs.

6 DISCUSSION

6.1 Threats to Validity

RQ1 and RQ3 are questions about approaches and algorithms, not implementations. However, as usual our evaluation can only compare representative implementations of the approaches. Implementing TreeLine and SlackLine atop AFL goes some way toward facilitating a meaningful comparison to PerfFuzz as the leading representative of the mutational approach, but it is imperfect. In particular, while we expect that simple byte-string mutation would generate mutants at a faster pace than our grammar-based search strategies even if the implementations were more evenly matched, we do not know precisely how much of the difference to attribute to a C implementation within the AFL process versus the Python implementation and client/server architecture of our grammar-based prototype tools.

A variety of other incidental implementation decisions might affect the results. There are, in short, a lot of knobs to adjust, and we have surely not found the optimum setting of every knob or combination.

All the approaches considered fundamentally depend on provided artifacts: a grammar in the case of TreeLine and SlackLine, and seed inputs in the case of PerfFuzz. We have tried to use "minimum effort" artifacts in the evaluation — a single example input as a seed for PerfFuzz where possible, and a grammar derived directly from documentation or converted from a parser specification file for grammar-based tools. In the case of `lunasvg` and `flex`, we provided PerfFuzz a set of seed inputs that included all the constructs in the grammar used by TreeLine and SlackLine. While we have tried to be fair (and in particular, not to "tune" grammars to improve grammar-based tools relative advantage), results could be different with a different set of grammars or seeds.

Comparisons of tools that require even a small amount of human configuration can be done only with a limited set of examples, which may or may not be representative of a larger population of applications. We have selected two examples (`wf` and `libxml2`) that were previously used in evaluation of the PerfFuzz approach, which we believe are representative of systems for which byte string mutation should be advantageous. Moreover, we introduced three new examples (`lunasvg`, `graphviz` and `flex`) which we believe are representative of applications driven by syntactically richer input, and therefore more favorable to the grammar-based approach. A larger and more varied set of examples would be desirable, but will take time to accumulate.

6.2 Alternatives

We measured and bounded the length of generated inputs in bytes to make a fair comparison to PerfFuzz. That is not the choice we would have made otherwise. Algorithmic efficiency is unlikely to be determined primarily by scanning the input; a higher level measure such as number of tokens is likely to be better.

Producing grammars for use with our prototype tool is considerably easier than writing a grammar for a parser. It does not have to be LALR(1) or LL(k) or even unambiguous. We found it short work to transcribe a Yacc parser grammar in `graphviz` to an acceptable form, and to compose a simple grammar for `lunasvg` and `flex` from a page of documentation. Nevertheless it might be better yet if we could infer grammars directly from program behavior. Preliminary experience with Glade [6] has not produced useful grammars. Experience with Arvada [21] has been mixed, and will be pursued farther. We have not yet tried white-box or grey-box grammar inference engines such as Mimid [16, 19].

7 CONCLUSIONS

We have demonstrated that grammar-based search is dramatically more effective than mutational fuzzing of byte strings for finding pathological inputs to applications whose behavior is directed by richly structured textual input. The main challenge was adapting techniques such as Monte Carlo Tree Search, typically tuned for particular games with 0/1 outcomes, to work well across varying applications with execution costs that range widely and unpredictably. TreeLine and SlackLine dynamically tunes their search parameters, including a reward function that adapts to the range of execution costs it encounters. They also produce a bound sized input that demonstrate a performance issue in the structure of the input rather than it scale.

DECLARATIONS

The version of TreeLine and SlackLine used in this paper is permanently available through Zenodo [2]. Also, we share the experimental data used in the plots through FigShare [3].

ACKNOWLEDGMENTS

We appreciate the foundation for further experimentation provided by the authors of AFL and PerfFuzz.

REFERENCES

[1] Ziyad Alsaeed. 2023. Performance issue in rendering files using sw_ft_raster.c #94. https://github.com/sammycage/lunasvg/issues/94. Accessed: May 2023.

[2] Ziyad Alsaeed and Michal Young. 2023. *Artifact: Finding Short Slow Inputs Faster with Grammar-Based Search.* https://doi.org/10.5281/zenodo.7970349

[3] Ziyad Alsaeed and Michal Young. 2023. Experimentation Data: Finding Short Slow Inputs Faster with Grammar-Based Search. (6 2023). https://doi.org/10.6084/m9.figshare.22114373.v1

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. https://doi.org/10.14722/ndss.2019.23412

[5] Hendrik Baier and Peter D. Drake. 2010. The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 4 (2010), 303–309. https://doi.org/10.1109/TCIAIG.2010.2100396

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (Sep 2017), 95–110. https://doi.org/10.1145/3140587.3062349

[7] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

[8] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Stanford, California) *(AIIDE'08)*. AAAI Press, 216–217.

[9] 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* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/2884781.2884794

[10] World Wide Web Consortium. 2023. W3C - Scalable Vector Graphics (SVG) 2: Paths. https://www.w3.org/TR/SVG/paths.html#PathElement. Accessed: Feb 2023.

[11] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games* (Turin, Italy) *(CG'06)*. Springer-Verlag, Berlin, Heidelberg, 72–83.

[12] Marcelo de Barros. 2021. wf - Simple Word Frequency Counter (version 0.41). http://ftp.altlinux.org/pub/distributions/ALTLinux/Sisyphus/x86_64/SRPMS.classic/wf-0.41-alt2.src.rpm. Accessed: Jan 2021.

[13] Ted Dunning. 2021. The t-digest: Efficient estimates of distributions. *Software Impacts* 7 (2021), 100049. https://doi.org/10.1016/j.simpa.2020.100049

[14] Ted Dunning and Otmar Ertl. 2019. Computing Extremely Accurate Quantiles Using t-Digests. arXiv:1902.04023 [stat.CO] https://arxiv.org/abs/1902.04023

[15] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. 2010. Fuego—An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 4 (2010), 259–270. https://doi.org/10.1109/TCIAIG.2010.2083662

[16] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 172–183. https://doi.org/10.1145/3368089.3409679

[17] 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* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 156–166. http://dl.acm.org/citation.cfm?id=2337223.2337242

[18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

[19] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 720–725.

[20] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based Monte-Carlo Planning. *IN: ECML-06. NUMBER 4212 IN LNCS* (2006), 282–293. https://doi.org/10.1.1.102.1296

[21] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 456–467.

[22] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[23] Vern Paxson. 2021. Flex - a scanner generator - Deficiencies. https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_node/flex_23.html. Accessed: Aug 2021.

[24] Vern Paxson. 2021. flex - The Fast Lexical Analyzer - scanner generator for lexing in C and C++ (version 2.6.4). https://github.com/westes/flex/archive/refs/tags/v2.6.4.tar.gz. Accessed: Aug 2021.

[25] Diego Perez, Spyridon Samothrakis, and Simon Lucas. 2014. Knowledge-based fast evolutionary MCTS for general video game playing. In *2014 IEEE Conference on Computational Intelligence and Games*. 1–8. https://doi.org/10.1109/CIG.2014.6932868

[26] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[27] AT&T Labs Research. 2021. GraphViz - Graph visualization software (version 2.47.0). https://gitlab.com/graphviz/graphviz/-/package_files/8183714/download. Accessed: Jul 2021.

[28] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume M. J. B. Chaslot, and Jos W. H. M. Uiterwijk. 2008. Single-Player Monte-Carlo Tree Search. In *Computers and Games*, H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

[29] 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* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 270–281. https://doi.org/10.1145/2771783.2771816

[30] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (jan 2016), 484–489. https://doi.org/10.1038/nature16961

[31] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/3460319.3464814

[32] Samuel Ugochukwu. 2022. LunaSVG - SVG rendering library in C++. https://github.com/sammycage/lunasvg. Accessed: Aug 2022.

[33] Daniel Veillard. 2021. Libxml2 - The XML C parser and toolkit of Gnome (version 2.9.7). http://xmlsoft.org/sources/libxml2-2.9.7-rc1.tar.gz. Accessed: June 2021.

[34] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

[35] Michał Zalewski. 2020. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2020-12-26.

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

Received 2023-02-16; accepted 2023-05-03