TreeLine and SlackLine: Grammar-Based Performance Fuzzing on Coffee Break

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

TreeLine and SlackLine are grammar-based fuzzers for quickly finding performance problems in programs driven by richly structured text that can be described by context-free grammar. In contrast to long fuzzing campaigns to find (mostly invalid) inputs that trigger security vulnerabilities, TreeLine and SlackLine are designed to search for performance problems in the space of valid inputs in minutes rather than hours. The TreeLine and SlackLine front-ends differ in search strategy (Monte Carlo Tree Search or derivation tree splicing, respectively) but accept the same grammar specifications and rely on a common back-end for instrumented execution. Separation of concerns should facilitate use by other researchers who wish to explore alternatives and extensions of either the front or back ends.

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. TreeLine and SlackLine: Grammar-Based Performance Fuzzing on Coffee Break. 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, 4 pages. https://doi.org/10.1145/3597926.3604925

1 INTRODUCTION

Consider a developer reimplementing or maintaining the Unix scanner generator `flex`. Following modern development practices, the developer will certainly contribute unit tests to check modified or added code, and re-execute existing test suites to check for unanticipated interactions. However, these unit tests are designed to check for functionality bugs. It is difficult to write unit test cases to check for performance bugs. Search-based test data generation ("fuzzing") is a more appropriate tool for finding inputs that trigger unanticipated performance problems.

TreeLine and SlackLine are experimental tools for grammar-based search to find textual inputs that trigger slow execution [2]. Like most fuzzing tools, they explore a space of possible inputs in search of a few that reveal potential bugs — in this case, performance bugs, indicated by unusually large counts of control flow edge executions triggered by inputs of strictly limited length. Unlike most fuzzing tools, TreeLine and SlackLine are designed for short runs, measured in minutes rather than hours or days. In minutes our imaginary maintainer of `flex` may use TreeLine or SlackLine to generate an example input such as the one shown below which produces a 36 megabyte output file.

```
([^Yq]*?\b.?*|[^t ]*?7.............*.|[^5Y]*?*)_ ;
```

TreeLine and SlackLine are robust enough for practical use, but their primary intended users are researchers exploring extensions and alternative approaches to performance fuzzing. They produce sample inputs with metadata in a form that can be easily scanned for comparison with alternative approaches. The code base is organized to facilitate creation and evaluation of search strategies atop the shared infrastructure of TreeLine and SlackLine, or configurable variants of those tools. The execution and instrumentation component of TreeLine and SlackLine is itself an extension of PerfFuzz [11], which in turn is an extension of AFL [17].

Considering this intended use by other researchers, we describe both basic use of existing tools and sketch how TreeLine and SlackLine can serve as a parts bin for building and evaluating other grammar-based performance fuzzers.

2 BACKGROUND

TreeLine and SlackLine draw from multiple lines of research. We note here some direct influences, reuse of tools, and an overview of the approaches to search used in TreeLine and SlackLine. Additional details can be found in the ISSTA paper [2].

Feedback guided fuzzing for performance. America Fuzzy Lop (AFL) [17] has been a foundation for many research tools in fuzzing. SlowFuzz [14] and then PerfFuzz [11] are built on AFL [17], adding performance feedback for multi-objective optimization. The instrumentation back-end of TreeLine and SlackLine is in turn built on AFL with PerfFuzz enhancements (but without using any part of the mutation facilities inherited by SlowFuzz and PerfFuzz).

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

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3604925

Grammar based fuzzing. Generating test cases from context-free grammars far predates modern feedback-guided fuzzers [9]. Among recent work on using grammars with fuzzers, SlackLine (but not TreeLine) borrows the approach of derivation tree splicing from Nautilus [5]. As the Nautilus tool is built in Rust and SlackLine is a Python application, we were not able to reuse any part of the Nautilus source code, but the SlackLine's "chunk store" (previously generated subtrees suitable for splicing into another derivation tree) is a Python reimplementation of Nautilus's chunk store. One notable difference in generation strategy between Nautilus and SlackLine is that we systematically and cheaply generate only derivation trees whose string representation is within a fixed length budget. This is essential for performance fuzzing, less so if fuzzing to find functional bugs and security vultnerabilities. The Nautilus authors have reported trying and abandoning a more expensive strategy for limiting generated string length.

Search. TreeLine and SlackLine implement two search strategies, both influenced by Monte Carlo Tree Search (MCTS). In TreeLine, the branches of the search tree are derivation steps (replacement of a non-terminal symbol by the right-hand side of a production), and leaves of the tree are complete derivations. TreeLine can be considered a true MCTS, balancing exploration with exploitation, but with a number of accomodations for differences between game search (with simple "win" or "lose" outcomes) and performance fuzzing. One of the crucial adjustments is converting control-flow edge hits from raw counts to quantiles using a t-digest [8].

For SlackLine, nodes in the search tree are complete derivation trees, most search steps are splices as in Nautilus (but subject to length bounds). While SlackLine propagates scores as in MCTS, it only roughly approximates the selection of a node to expand in MCTS, instead (optionally) using node scores to select higher scoring derivations more often than lower scoring derivations.

The search approaches of TreeLine and SlackLine are illustrated in Figure 1, both based on the trivial grammar at the top (Figure 1a). We assume we work with a budget of 2.

The cost of a derivation step is the difference between the minimum length of a string generated by a non-terminal symbol and the minimum length of a string generated by the right-hand side of the selected rule. Calculating the costs for the grammar we would have a cost of 1 for rules ① and ② and 0 for rule ③.

Figure 1b illustrate a "rollout" step in TreeLine. Nodes in the search tree represent phrases, so a path from root to leaf represents a (partial) derivation sequence. Rollouts may be a sequence of derivation steps, as in the case of the leftmost path of D_i from $\langle S \rangle$ to xx by applying grammar rule ① twice and then grammar rule ③. Note that we cannot use any other rule than ③ after choosing rule ① twice as we consumed the full budget. If the node representing Sy is selected (as in D_{i+1}), we may apply grammar rule ③ to obtain the string y. When one of the leaf nodes has been thoroughly explored (currently, when we have executed 20 rollouts from that node), we may instead perform an "expand" operation, which populate its children.

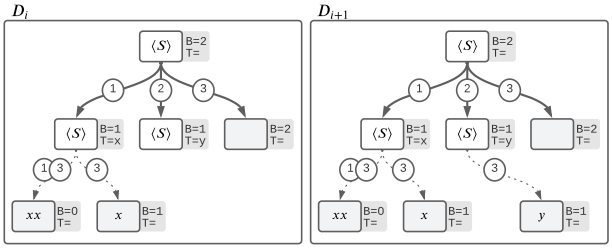
In SlackLine or SlackLineMC, derivation trees are hybridized by replacing previously generated subtrees. Figure 1c illustrates generation of derivation tree D_3 for the string "yx" can be obtained by substituting derivation tree D_1 for the left subtree of the root

① ⟨S⟩ ::= ⟨S⟩ 'x'

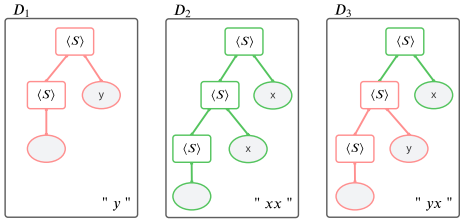
② ⟨S⟩ ::= ⟨S⟩ 'y'

③ ⟨S⟩ ::= /* empty */

(a) Simple Grammar



(b) Possible TreeLine based derivations.



(c) Possible SlackLine based derivations.

Figure 1: Search steps in TreeLine (Figure 1b) and SlackLine (Figure 1c). Both based on the tiny grammar at the top (Figure 1a). In both derivation steps and substitutions are constrained by a length limit on generated strings.

in derivation tree D_2. The frontier of the search is a collection of derivation trees that have scored highly using the same scoring criteria as in TreeLine. When a new derivation tree is added to the frontier of the search, at the same time all of its subtrees are added to a "chunk store", from which substitutions may be selected. Derivation trees on the frontier may be selected in a breadth first order (SlackLine) or a weighted selection that roughly approximates MCTS (SlackLineMC) by setting a parameter in the configuration file. Similar to TreeLine, derivations and substitutions are constrained by the allowed budget.

3 USING TREELINE AND SLACKLINE

TreeLine [4] and SlackLine [3] are each packaged as a GitHub repository with a build script that produces a Docker container built on Debian Linux. The Docker container runs two processes, a "front end" search strategy component (either TreeLine or SlackLine) and a "back end" server that executes the instrumented application, as illustrated in Figure 2.

The user provides a target application in source form, along with a context free grammar to direct generation of test inputs. The target applications accepted by TreeLine and SlackLine are those accepted by AFL with LLVM instrumentation. Example applications from the ISSTA paper [2] are included in the repository.

The user also provides a context-free grammar to guide input generation, expressed in extended BNF. Each of the provided target applications is accompanied by at least one grammar. Sometimes

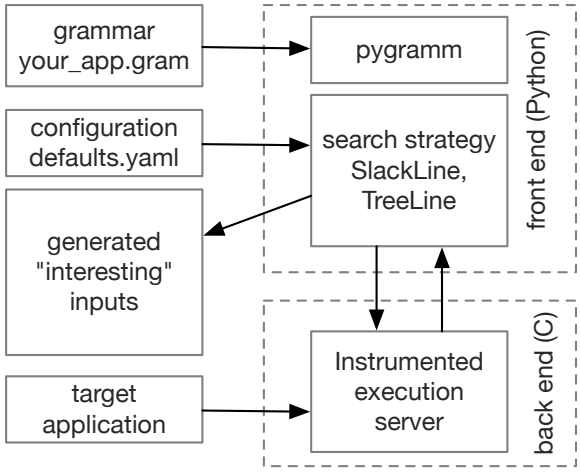


Figure 2: Structure of TreeLine and SlackLine. The user provides a grammar, a configuration file, and C or C++ source code for a target application. The target application is instrumented and run repeatedly in an instrumented execution server, responding to requests from a search strategy module in a separate process, all within a Docker container.

such a grammar can be easily taken (almost transcribed) from application documentation, as we did for `flex`. Sometimes a parser specification written in EBNF, as for Yacc/Bison or Antlr, can be lightly edited into the format accepted by our grammar component `pygramm`; this is the approach we took with `graphviz`. Sometimes a simple grammar must be written from scratch.

The grammar need not be precise. It is enough that it produces a fairly high proportion of valid inputs. It also need not be comprehensive. A grammar that does not produce all valid inputs is especially useful in two cases. Sometimes it is desirable to intentionally omit some valid inputs with known performance impacts, to search for other, unexpected performance issues. We took this approach with `flex`, purposely omitting the dangerous trailing context feature after discovering that in addition to documented performance impacts on generated scanners [13], it impacted scanner *generation* performance as well. This led us to find other inputs that blew up the size of generated scanners and the performance of scanner generation with `flex`. A grammar that is not comprehensive may also be much simpler than a comprehensive grammar, and may generate fewer useless variations. For example, if generating input that includes identifiers, a few sample identifiers may guide a more effective search than a more comprehensive grammar that can produce all legal identifiers. We used this approach to build a simple SVG grammar for `lunasvg`. (The authors of Nautilus have also used and recommended this tactic.)

Although it is also possible to synthesize a grammar automatically [6, 10], in our experience even a sloppy, quick-and-dirty handwritten grammar is usually superior. If application input does not have a structure that is reasonably easy to describe with a context-free grammar, TreeLine and SlackLine are unlikely to outperform fuzzers that mutate unstructured strings of bytes.

The user can also vary fuzzing options with a configuration file written in YAML. Most notable is the limit on the length of generated inputs. We provide a standard option named "`budget`" that could mean the number of bytes or tokens (literal strings in the grammar). Measurements in the accompanying ISSTA paper were all made with budget limits in bytes for a fair comparison to tools like PerfFuzz that mutate byte strings, but we believe tokens are a more appropriate unit for grammar-based test case generation.

Other key options include the "`gram_file`", a path to the grammar file that guides input generation, and "`time`", the duration of the fuzzing run. Many other and varying options are documented in a `defaults.yaml` file for each tool.

TreeLine and SlackLine record inputs that increase some metric (coverage, hot spot, or total cost) as text files in a directory containing the configuration and results of a single run of the fuzzer. Metadata about the generated input, including the time of generation, the reason an input was saved, and its cost, is encoded in the file name, to enable simple post-processing by shell scripts.

The process of building, using, and interpreting results of TreeLine and SlackLine (essentially identical except for available configuration settings) is illustrated in an accompanying video at https://youtu.be/zVFKtiIhYUA.

4 RESULTS

Grammar-based fuzzing with tools like TreeLine and SlackLine is suited to applications whose functionality responds to textual input with rich syntactic structure. It is not suited to binary file formats, which binary fuzzers like AFL and PerfFuzz can explore much more quickly, nor is it advantageous if the most expensive inputs are syntactically malformed (e.g., a parser). The strength of grammar-based fuzzing is in more efficiently exploring how syntactic features of input control semantic processing, or what Aschermannn *et al.* call "fishing for deep bugs" [5].

We illustrate the performance advantage of grammar-based performance fuzzing with three suitable open source applications, all written in C or C++.

- `graphviz-2.47.0` [15] is a widely used graph visualization toolkit, of which the most widely used tool is *dot*, which lays out a directed graph from a graph description in a notation also called *dot*. We extracted an input grammar for `graphviz` *dot* by stripping extraneous code from the parser packaged with `graphviz`.
- `flex-2.6.4` [12] is a very widely used generator of lexical analyzers (scanners). We extracted a grammar for `flex` from its documentation pages. After discovering that the documented "dangerous trailing context" feature of `flex` not only slows down the generated lexers (as documented), but also slows `flex` itself, we prepared a more restricted grammar that omits that feature, titled "flex-limited" in the graphs. Even without dangerous trailing context, SlackLine and SlackLineMC quickly find short inputs that cause `flex` to generate enormous scanners.
- `lunasvg-2.3.2` [16] is a popular standalone library for rendering scalable vector graphics (SVG), a standard XML-based notation supported by most web browsers and many illustration applications including the Adobe suite. We composed a partial SVG grammar from W3C [7] documentation with a focus on the path element. In the course of evaluation we discovered a hang bug which we reported to the `lunasvg` developers [1], who repaired it by replacing a font rendering library.

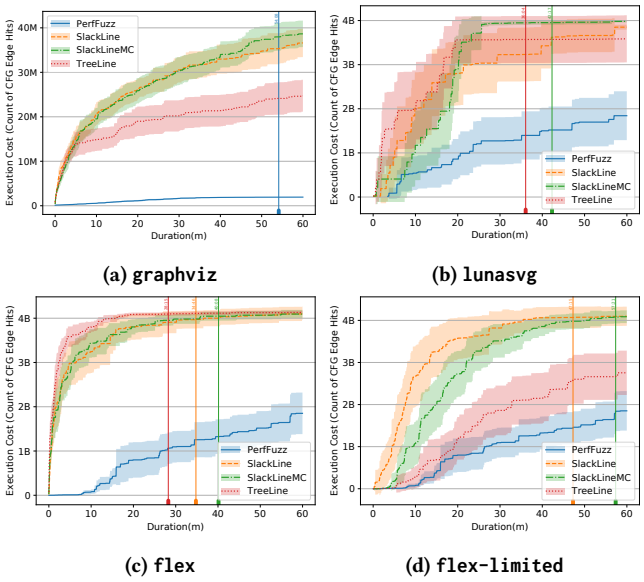


Figure 3: A comparison between PerfFuzz, SlackLine, SlackLineMC, and TreeLine showing maximum path length found in 60 minutes. Lines are averages and bands show 95% confidence interval across 20 repetitions.

Figure 3 illustrates the performance of TreeLine and of SlackLine with and without weighted selection of frontier nodes based on Monte Carlo Tree Search scoring. They can be compared to the performance of PerfFuzz, which uses the same instrumentation, but relies on conventional genetic search while mutating binary strings. All three tools use a count of control flow edge executions as a proxy for algorithmic complexity. The data presented in Figure 3 is excerpted from evaluation data presented in more detail in the accompanying ISSTA paper [2].

While each evaluation ran for 60 minutes and was repeated 20 times to obtain 95% confidence intervals on performance, from a practical viewpoint it is useful to observe how steeply the curves in Figure 3 rise in the first 20 minutes. Although difficult to quantify, our observation is that if there is a pattern that triggers exceptionally slow execution, TreeLine and SlackLine usually find it within the first 20 minutes, and thereafter refine it. This is what we mean by "performance fuzzing on coffee break."

5 EXTENDING THE TOOLS

TreeLine and SlackLine implement two search strategies, but share much infrastructure. They are structured for further experimentation, extending or replacing any of the major components.

The grammar processing component `pygramm` pre-processes a grammar to associate a minimum length with each symbol (terminal and non-terminal) in the input grammar. It can then efficiently generate only sentences up to a length limit (budget). Literals may be assigned length 1 (if the unit of length is tokens) or the length of that literal in bytes (if the unit of length is bytes). The fixpoint calculation that propagates minimum lengths to non-terminal symbols is simple, like an iterative data flow analysis, but we are not aware of other grammar-based generators that provide it.

A search strategy inevitably requires dozens of design decisions, which interact in many ways. Many rounds of informal evaluation of alternatives during development is essential, but still unlikely to have produced the very best combination of settings and features, let alone thorough understanding of all the interactions. Many features and variations are settable through the configuration file.

Python accelerates the pace at which we and others can develop and evaluate search strategies, but the AFL tool that we depend on to instrument and execute target applications is written in C. We therefore wrapped the application runner, based on AFL as extended in PerfFuzz, in a server that runs in a separate Unix process. This serverized version of AFL adds communication and synchronization overhead, but significantly eases the task of integrating a new search strategy component, or potentially of substituting a different engine for instrumented execution.

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. Finding Short Slow Inputs Faster with Grammar-Based Search. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, United States) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598118
[3] Ziyad Alsaeed and Michal Young. 2023. SlackLine: A grammar-based mutational fuzzing tool to generate performance testing inputs. https://github.com/uo-se-research/slackline.git. Accessed: May 18, 2023.
[4] Ziyad Alsaeed and Michal Young. 2023. TreeLine: Finding Slow Inputs Faster with Monte-Carlo Tree Search. https://github.com/uo-se-research/treeline.git. Accessed: May 18, 2023.
[5] 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
[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] World Wide Web Consortium. 2023. W3C - Scalable Vector Graphics (SVG) 2: Paths. https://www.w3.org/TR/SVG/paths.html#PathElement. Accessed: Feb 2023.
[8] Ted Dunning. 2021. The t-digest: Efficient estimates of distributions. *Software Impacts* 7 (2021), 100049. https://doi.org/10.1016/j.simpa.2020.100049
[9] Kenneth V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 4 (1970), 242–257.
[10] 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.
[11] 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
[12] 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.
[13] Vern Paxson. 2023. Flex - a scanner generator - Deficiencies. https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_node/flex_23.html. Accessed: May 2023.
[14] 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
[15] 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.
[16] Samuel Ugochukwu. 2022. LunaSVG - SVG rendering library in C++. https://github.com/sammycage/lunasvg. Accessed: Aug 2022.
[17] Michał Zalewski. 2020. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2020-12-26.

Received 2023-05-18; accepted 2023-06-08