Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques

Yuanmin Xie[*]
School of Software, KLISS, BNRist
Tsinghua University
China
xieym23@mails.tsinghua.edu.cn

Zhenyang Xu[*]
School of Computer Science
University of Waterloo
Canada
zhenyang.xu@uwaterloo.ca

Yongqiang Tian
Monash University
Australia
yongqiang.tian@monash.edu

Min Zhou
School of Software, KLISS, BNRist
Tsinghua University
China
mzhou@tsinghua.edu.cn

Xintong Zhou
School of Computer Science
University of Waterloo
Canada
x27zhou@uwaterloo.ca

Chengnian Sun
School of Computer Science
University of Waterloo
Canada
cnsun@uwaterloo.ca

ABSTRACT

Compiler testing is critical and indispensable to improve the correctness of compilers. Spurred by recent advancements in Large Language Models (LLMs), LLM-based compiler testing techniques such as Fuzz4All, have demonstrated their potential in uncovering real bugs in diverse compilers and reducing the required engineering efforts in designing program generators. Given the continuous evolution of LLMs and the emergence of new LLM-based approaches, establishing robust baselines is crucial for rigorous evaluation and driving future advancements in this promising research direction.

To this end, we introduce Kitten, a mutation-based, language-agnostic program generator. Kitten leverages a corpus of seed programs, analogous to the training set for LLMs, and utilizes the target language's syntax, akin to the knowledge learned by LLMs. Furthermore, Kitten's mutation operators can generate diverse test programs, demonstrating a behavior analogous to the ability of LLM inference to generate new code.

Our evaluations demonstrate that, using existing compiler test suites as seed programs, Kitten outperforms Fuzz4All in terms of code coverage and bug detection capabilities. Within 24 hours, Kitten achieved 48.3%, 9.9%, and 33.8% higher coverage than Fuzz4All on GCC, LLVM, and Rustc, respectively, while identifying an average of 19.3, 20.3, and 15.7 bugs in these compilers across three runs. Over the course of nine months dedicated to Kitten's development and testing, we identified a total of 328 across the compilers GCC, LLVM, Rustc, Solc, JerryScript, scalac, and slang, of which 310 have been confirmed or fixed. We strongly believe that Kitten serves as an effective baseline, enabling the identification of limitations within existing LLM-based approaches and consequently driving advancements in this promising research direction.

[*] Both authors contributed equally to this research.

ISSTA Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1474-0/2025/06
https://doi.org/10.1145/3713081.3731731

CCS Concepts

• **Software and its engineering → Compilers**; **Software testing and debugging**.

KEYWORDS

Compiler Testing, Language-Agnostic Code Generation, Benchmarking

ACM Reference Format:
Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. 2025. Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques. In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25), June 25–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3713081.3731731

1 INTRODUCTION

Ensuring compiler correctness is paramount for the reliability of all software systems, making compiler testing a critical and active research area. Traditional approaches, which rely on manually engineered random program generators to generate test programs for specific languages [18, 30], have discovered numerous compiler bugs. The recent surge in Large Language Model (LLMs) capabilities has opened new avenues for automated test program generation. Trained on massive datasets, LLMs exhibit a strong understanding of programming languages, enabling the generation of diverse test programs tailored to different compilers by prompting LLMs. Recent LLM-based approaches [28, 29] demonstrate the potential of using LLMs to generate random test programs for compiler testing. Compared to traditional methods, LLM-based approaches offer a significant advantage of language-agnosticism, enabling diverse test program generation across various programming languages, and substantially reducing the engineering effort required for developing specialized test program generation tools.

Prior work, such as Fuzz4All [28], has demonstrated promising results in compiler testing, achieving higher code coverage than traditional methods (*e.g.*, Csmith [30], YARPGen [18], and GrayC [6]). We believe that the continuous evolution of LLMs will further enhance the effectiveness of these approaches. However, as pointed out in [26], using traditional methods—especially language-specific program generators such as Csmith, YARPGen and GrayC—as evaluation baselines is subject to bias. These baselines were often designed for different purposes: Csmith and GrayC, for example, focus on generating well-defined programs with respect to the language specification to find correctness bugs in compiler optimizations, limiting the diversity of generated tests. In contrast, LLM-based approaches like Fuzz4All aim to find compiler crashes and hangs, potentially generating programs that are not well-defined or even compilable. Due to these inherent differences in their objectives, Fuzz4All was able to reach more areas of the compiler than the traditional approaches.

To rigorously evaluate and guide future research on LLM-based compiler testing, we propose using Kitten[1] as a new baseline. Kitten shares great similarities with LLM-based approaches. Both aim to find crashes and hangs in compilers and thus do not guarantee semantic validity of the generated tests. Second, Kitten is mutation-based and requires a corpus of seed programs, analogous to the training set for LLMs, and utilizes the target language's syntax, akin to the partial knowledge learned by LLMs. Kitten's mutation operators serve as an inference mechanism to generate new test programs by leveraging the seed programs and the language syntax, demonstrating a behavior analogous to LLM inference.

Our experiments show that Kitten outperforms Fuzz4All in both code coverage and bug detection. Over 24 hours, Kitten achieved 48.3%, 9.9%, and 33.8% higher coverage on GCC, LLVM, and Rustc, respectively. Beyond coverage, Kitten also performed well in bug detection, identifying an average of 19.3, 20.3, and 15.7 bugs in GCC, LLVM, and Rustc across three runs. Additionally, our results confirm that Kitten achieves better results than the state-of-the-art grammar-based fuzzer Grammarinator [9], reinforcing its effectiveness in compiler testing. The results of our comparative experiments highlights Kitten's ability to uncover issues across diverse compilers. Since its development, Kitten has discovered a total of 328 bugs across multiple compilers for the languages of C, Rust, Solidity, JS, Scala and Verilog. Among these, 310 have already been confirmed or fixed.

We strongly believe that Kitten serves as an effective baseline, enabling the identification of limitations and improvement opportunities within existing LLMs-based approaches and consequently driving advancements in this promising research direction.

Contributions. We make the following main contributions.

(1) A language-agnostic compiler testing tool, Kitten, serves as a simple yet effective baseline for LLM-based methods.
(2) A comprehensive experiment comparing Kitten and Fuzz4All shows that Kitten achieves 48.3%, 9.9%, and 33.8% higher coverage on GCC, LLVM, and Rustc, respectively, compared to Fuzz4All. Kitten also identified an average of 19.3 bugs on GCC, 20.3 bugs on LLVM, and 15.7 bugs on Rustc. Additionally, Kitten outperformed the grammar-based fuzzer Grammarinator, further demonstrating its effectiveness.

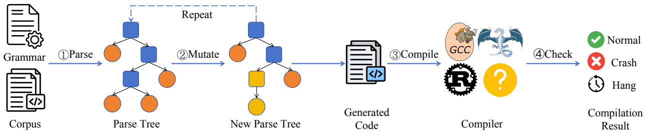


Figure 1: Workflow of Kitten

(3) Kitten detected a total of 328 bugs across various compilers, including GCC, LLVM, Rustc, Solc, JerryScript, scalac, and slang.

2 DESIGN

Kitten is a language-agnostic program generator, designed for detecting bugs in various compilers. Figure 1 illustrates the workflow of Kitten. Kitten takes as input a corpus of seed programs and the grammar file of the target language specified in the Antlr format [20]. Kitten achieves its language-agnosticism by exploiting the syntactical knowledge encoded in the grammar file, to build parse trees from seed programs and to perform syntax-guided mutations on top of the parse trees. Kitten's reliance on the Antlr format for language syntax enables broad language support for compiler testing. Antlr possesses a large user community. With the contribution from its user community, the grammars of many prevalent programming languages are contributed and readily available within the Antlr grammar repository [1], which consists of 546 grammar files for 297 languages up to Jan 2025.

Specifically, the workflow of Kitten includes the following steps:

Step 1: Parse. Kitten uses the provided Antlr grammar to parse each seed program into a parse tree. Each parse tree contains two types of nodes: non-leaf nodes which represent non-terminal symbols and leaf nodes which represent tokens.

Step 2: Mutate. Inspired by syntax-guided program reduction [25], Kitten can leverage the syntactical knowledge encoded in the grammar to perform mutations, which ensures the generation of syntactically valid test programs. Kitten can also generate syntactically invalid test programs to stress test compilers with invalid programs.

Step 3: Compile. The parse trees can be easily converted back into programs, and then are fed to the compiler for compilation.

Step 4: Check. Kitten uses the abnormal behaviors of the compiler, including crashes and hangs, as oracles to detect bugs.

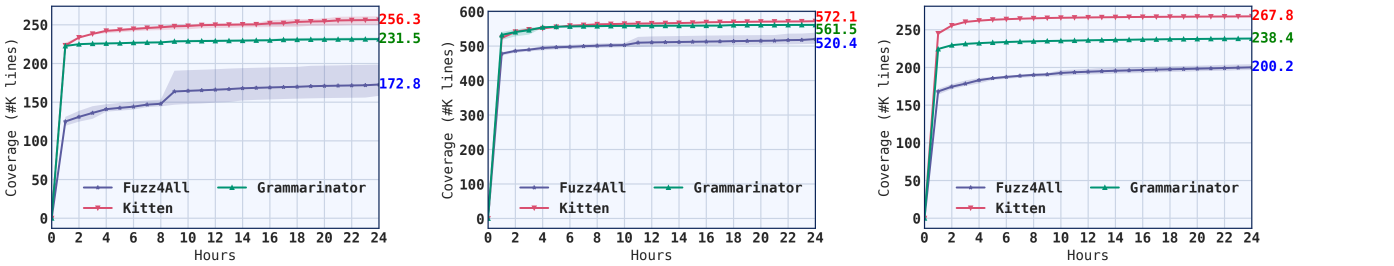
2.1 KEY FEATURES

2.1.1 Mutation. Kitten performs mutations at two levels: tree-level and token-level.

Tree-Level Mutations. Tree-level mutations are applied to the parse trees, ensuring that modifications maintain syntactic validity. Inspired by NAUTILUS [2], Kitten employs the following tree-level mutation strategies:

- **Splicing:** This mutation involves two parse trees. A subtree from one parse tree is selected and used to replace a subtree in another parse tree, resulting in a new tree.
- **Replace:** This mutation replaces a subtree in the parse tree with a new one, generated based on the grammatical rules compatible with the original subtree.

[1] Kitten was first proposed in a preliminary, full research paper [26] to empirically demonstrate the challenges in evaluating machine learning-based compiler testing techniques. This paper presents the full technical details of Kitten together with comprehensive new evaluation results, These results include the discovery of a total of 328 bugs across various compilers, and a detailed comparison study with a novel LLM-based compiler testing technique, Fuzz4All.



(a) GCC **(b) LLVM** **(c) Rustc**

Figure 2: Line coverage by Kitten, Fuzz4All and Grammarinator.

- **Delete:** This mutation removes repetitive structures in the parse tree, such as lists and loops, which follow grammatical rules that allow multiple repetitions.
- **Repeat:** This mutation generates a recursive subtree on a non-leaf node in the parse tree. Unlike **Replace**, the generated subtree exhibits a clear recursive structure.

Token-Level Mutations. Token-level mutations operate on the token sequence derived from the parse tree, encompassing three types: *insertion*, *deletion*, and *replacement*. Unlike tree-level mutations, these operators do not preserve syntactic validity, which makes them effective for identifying bugs in the syntax analysis components of compilers.

2.1.2 Execution Modes. Kitten supports two execution modes for different testing needs. The first generates programs and feeds them to the compiler on the fly, focusing on rapid bug detection and retaining only inputs that trigger bugs, along with reproduction scripts. The second stores all generated programs without requiring a specific compiler, enabling later analyses such as coverage measurement.

2.1.3 Multi-threading and Bug Management. Kitten includes practical engineering features to enhance usability and efficiency. It provides multithreaded code generation to maximize CPU utilization. Further, The bug management of Kitten, including auto-reduction and bug deduplication, save labor time by simplifying complex bug-triggering programs and eliminating duplicates automatically.

3 EVALUATION

To establish Kitten as a suitable baseline for evaluating the effectiveness of LLM-based techniques, we compare Kitten with Fuzz4All and the grammar-based fuzzer Grammarinator to examine the following questions.

Code Coverage: How do Kitten compare to the other two tools in terms of their overall code coverage and its distribution?

Bug Detection: How does Kitten compare to the other two tools in terms of the number of discovered bugs?

We selected two programming languages, C and Rust, to evaluate testing effectiveness, targeting their respective compilers, GCC 13.1.0, LLVM 19.1.6 and Rustc (commit `788202a`). We utilized code from each compiler's test suite as seeds for Kitten and Grammarinator, carefully excluding test cases that inherently cause compiler crashes or hangs to prevent redundant and meaningless bug reports. Since Fuzz4All supports testing across multiple compilers, we extended its functionality by implementing a Rust adapter, enabling it to generate test cases for Rust. In our experiments, we used the default configuration from Fuzz4All's public repository. To mitigate the effects of randomness and ensure a fair comparison, we conducted three independent runs for each tool under the same configuration.

3.1 RQ1: Code Coverage

Figure 2 illustrates the code coverage trends over 24 hours, comparing Kitten with other tools. The solid lines represent the average coverage, while the shaded areas depict the range of coverage variation. Among them, only Fuzz4All exhibits significant fluctuations. Overall, Kitten achieves the highest code coverage among these tools, particularly surpassing Fuzz4All by 48.3% on GCC, 9.9% on LLVM, and 33.8% on Rustc.

Table 1: Line coverage of different components of GCC.

	Front end	Middle& Back end	Other
Kitten	**50,623**	**99,628**	**106,065**
Fuzz4All	40,432	58,851	73,548
Grammarinator	45,481	87,484	98,550

Unlike traditional methods, LLM-generated code may fail to pass language parsing, potentially increasing coverage in the compiler front end. To investigate this, we analyzed coverage across the front end, middle/back end, and other components of GCC. As shown in Table 1, Fuzz4All does not outperform Kitten and Grammarinator in any specific, suggesting that the programs generated using LLMs fail to comprehensively stress certain parts of compilers, especially the middle/back end, which can be improved by future studies.

3.2 RQ2: Bug Finding Capability

Table 2: Number of unique bugs detected in three runs over 24 hours.

	GCC				LLVM				Rustc			
Tool	1	2	3	Avg	1	2	3	Avg	1	2	3	Avg
Kitten	19	21	18	19.3	22	23	16	20.3	13	16	18	15.7
Fuzz4All	4	6	7	5.7	0	1	0	0.3	0	0	0	0.0
Grammarinator	13	11	9	11.0	6	6	3	5.0	5	3	3	3.7

The number of bugs detected is a critical metric for evaluating compiler testing effectiveness. To avoid redundant results caused by duplicate bugs, we used Kitten's builtin bug management feature to deduplicate them based on the call stacks of the compiler, obtaining the unique bugs as shown in Table 2.

In three 24-hour experiments, Kitten detected an average of 19.3, 20.3, and 15.7 bugs in GCC, LLVM, and Rustc, respectively. Fuzz4All found 5.7 and 0.3 bugs[2] in GCC and LLVM but did not detect any bugs in Rustc. The grammar-based Grammarinator also detected more bugs than Fuzz4All. We re-tested the bugs detected by Kitten on the latest versions of the compilers and submitted 35 reports for those that could still be triggered, of which 17 have already been confirmed. Most GCC and Rustc bugs reported by Kitten were quickly confirmed by developers, whereas fewer LLVM bugs were confirmed, consistent with the slower confirmation pace noted in survey [24]. This result demonstrates the bug detection ability of Kitten on the latest compilers.

Table 3: All bugs detected by Kitten in seven compilers over nine months.

Compiler	GCC (C)	LLVM (C)	Rustc (Rust)	Solc (Solidity)	JerryScript (JS)	scalac (Scala)	slang (Verilog)
Bug Count	84	100	103	6	8	14	13

We also tested various compilers of diverse programming languages using Kitten. Over a total of nine months, Kitten detected 328 deduplicated bugs on seven compilers, as shown in Table 3. These results strongly demonstrate the effectiveness of Kitten in detecting compiler bugs across languages.

4 USAGE EXAMPLE

Kitten is intended for researchers and developers working on compiler testing. To use Kitten, a YAML configuration file is needed. This file specifies the seed set, the compiler under test, compilation flags, and the oracle used for validation.

```
# config.yaml
language: "C"
seedFolders:
  - path: "/path/to/seed"
    fileExtentions: [".c"]
programsUnderTest:
  - command: "/path/to/gcc"
    flagsToTest:
      - flags: ["-x", "c", "-std=c2x", "-c"]
    crashDetectorClassName: "GccCrashDetector"
```

Next, run the following command to start testing with the specified configuration, execution time, and mutation options.

```
java -jar kitten_deploy.jar \
  --testing-config config.yaml --timeout 3600 \
  --enable-splicing true --fuzzer-mode NORMAL_FUZZING
```

After execution, use the next command to organize detected bugs into directories and remove duplicates. Finally, the report of unique bugs is stored in the output directory.

```
java -jar kitten_organizer_deploy.jar --lang C \
  --delete-duplicates true
```

5 RELATED WORK

To test compiler correctness, numerous approaches to generating test programs have been proposed. Csmith [30] and its various variants [5, 15, 21] utilize language subsets or templates to generate test programs. NAUTILUS [2], Gramatron [22], Grammarinator [9] and EvoGFuzz [4] generate test cases based on a given grammar and perform mutations on them. They are closely related to Kitten. NAUTILUS and Gramatron are coverage-guided fuzzing techniques that do not rely on an externally provided seed corpus, whereas Grammarinator and EvoGFuzz are grammar-based fuzzers that benefit from an initial corpus. Kitten not only leverages the corpus but also employs mutation strategies designed for compiler testing, closely mimicking the behavior of LLM-based approaches. This establishes Kitten as an effective black-box baseline for evaluating LLM-based methods. In addition to grammar-based generation, mutation-based generation [6, 7, 16, 27] is also employed. Designing mutation strategies is not straightforward. EMI-based mutation [10–12, 23] applies semantics-preserving transformations to detect miscompilation. These approaches have uncovered numerous bugs in compilers for specific languages, but writing generation rules or mutation operators for each language requires significant manual effort. Beyond DL-based techniques [3, 13, 17], recent LLM-based approaches [8, 28, 29] leverage large language models [14, 19] to generate diverse, language-agnostic code, significantly reducing engineering effort. As a continuation of prior work [26], our study further evaluates LLM-based approaches and presents Kitten as a reproducible, language-agnostic baseline.

6 TOOL AVAILABILITY

For reproducibility and replicability, we have open-sourced Kitten at https://github.com/uw-pluverse/perses/tree/master/kitten and made all artifacts publicly available at https://doi.org/10.5281/zenodo.15044228. A demo video showcasing its usage is available at: https://youtu.be/hVqZBxRTr_4.

7 CONCLUSION

The emergence of LLMs has introduced new avenues for generating test programs; however, using traditional methods as baselines can lead to unfair comparisons. Thus, we propose Kitten as a baseline for LLM-based compiler testing tools. Kitten is a language-agnostic compiler testing tool that generates new test programs by mutating an existing corpus. Experimental results show that Kitten outperforms Fuzz4All in coverage on GCC, LLVM and Rustc, achieving 48.3%, 9.9% and 33.8% higher coverage, respectively. Moreover, during the experiments, Kitten identified 19.3 bugs in GCC, 20.3 bugs in LLVM, and 15.7 bugs in Rustc. Since its development, Kitten has identified a total of 328 bugs across various compilers. Based on these evaluations, we strongly believe that Kitten serves as an effective baseline for LLM-based compiler testing tools. Kitten, as well as the evaluation scripts, is open sourced to benefit future research.

ACKNOWLEDGMENTS

The authors from Tsinghua University were supported in part by the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92267203) and the National Key Research and Development Program of China (2022YFB43012). The authors from the University of Waterloo were supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Discovery Grant and by CFI-JELF Project #40736.

[2] Note that the number of Fuzz4All-found bugs in this paper is less than that reported in the original paper. This is because we used 24 hours whereas the Fuzz4All paper did not mention how much time was used for bug finding, and its reported bugs were in C++ rather than C.

REFERENCES

[1] Antlr. 2024. Antlr Grammar Repository. Retrieved December 31, 2024 from https://github.com/antlr/grammars-v4

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[3] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 95–105. https://doi.org/10.1145/3213846.3213848

[4] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12420)*, Aldeida Aleti and Annibale Panichella (Eds.). Springer, 105–120. https://doi.org/10.1007/978-3-030-59762-7_8

[5] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative About Undefined Behaviour. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1219–1223. https://doi.org/10.1145/3324884.3418933

[6] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1219–1231. https://doi.org/10.1145/3597926.3598130

[7] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making no-fuss compiler fuzzing effective. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 194–204. https://doi.org/10.1145/3497776.3517765

[8] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2201–2203. https://doi.org/10.1145/3611643.3617850

[9] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 45–48. https://doi.org/10.1145/3278186.3278193

[10] Bo Jiang, Xiaoyan Wang, Wing Kwong Chan, T. H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. 2020. CUDAsmith: A Fuzzer for CUDA Compilers. In *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*. IEEE, 861–871. https://doi.org/10.1109/COMPSAC48688.2020.0-156

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[12] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

[13] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2613–2630. https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung

[14] Raymond Li et al. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=KoFOg41haE

[15] Jiawei Liu et al. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 530–543. https://doi.org/10.1145/3575693.3575707

[16] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–26. https://doi.org/10.1145/3527317

[17] Xiao Liu et al. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 1044–1051. https://doi.org/10.1609/AAAI.V33I01.33011044

[18] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

[19] OpenAI. 2023. GPT-4 Technical Report. https://openai.com/research/gpt-4. Accessed: 2024-12-11.

[20] Terence Parr, Sam Harwell, et al. 2024. ANTLR (ANother Tool for Language Recognition). https://www.antlr.org/ Accessed: 2024-12-09.

[21] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1483–1486. https://doi.org/10.1145/3597926.3604919

[22] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 244–256. https://doi.org/10.1145/3460319.3464814

[23] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[24] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 294–305. https://doi.org/10.1145/2931037.2931074

[25] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. https://doi.org/10.1145/3180155.3180236

[26] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 4873–4882. https://doi.org/10.24963/IJCAI.2023/542

[27] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. 2023. SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1062–1074. https://doi.org/10.1145/3611643.3616277

[28] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 126:1–126:13. https://doi.org/10.1145/3597503.3639121

[29] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 709–735. https://doi.org/10.1145/3689736

[30] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532