Revisiting the Evaluation of Deep Learning-Based Compiler Testing

Yongqiang Tian[1,2] , **Zhenyang Xu**[1] , **Yiwen Dong**[1] , **Chengnian Sun**[1] and **Shing-Chi Cheung**[2]
[1]University of Waterloo
[2]The Hong Kong University of Science and Technology
{yongqiang.tian, zhenyang.xu, yiwen.dong, cnsun}@uwaterloo.ca, scc.ust.hk

Abstract

A high-quality program generator is essential to effective automated compiler testing. Engineering such a program generator is difficult, time-consuming, and specific to the language under testing, thus requiring tremendous efforts from human experts with *language-specific* domain knowledge. To avoid repeatedly writing program generators for different languages, researchers recently proposed a *language-agnostic* approach based on deep learning techniques to automatically learn a program generator (referred to as DLG) from existing programs. Evaluations show that DLGs outperform Language-Specific Program Generators (LSGs) in testing compilers.

However, we argue that it is unfair to use LSGs as baselines to evaluate DLGs. LSGs aim to validate compiler optimizations by only generating compilable, well-defined test programs; this restriction inevitably impairs the diversity of the language features used in the generated programs. In contrast, DLGs do not aim to validate the correctness of compiler optimizations, and its generated programs are not guaranteed to be well-defined or even compilable. Therefore, it is not surprising that DLG-generated programs are more diverse in terms of used language features than LSG-generated ones.

This study revisits the evaluation of DLGs, and proposes a new, fair, simple yet strong baseline named Kitten for evaluating DLGs. Given a dataset consisting of human-written programs, instead of using deep learning techniques to learn a program generator, Kitten directly derives new programs by mutating the programs in the dataset. Extensive experiments with more than 1,500 CPU-hours demonstrate that the state-of-the-art DLGs fail to compete against such a simple baseline: 3 *v.s.* 1,750 hang bugs, 1 *v.s.* 34 distinct compiler crashes. We believe that DLGs still have a large room for improvement.

1 Introduction

Compilers are among the most important, fundamental system software. Every program, no matter whether it is an operating system or application, has to be compiled by a compiler from source code into binary executable, so that the program can be executed by a computer. Hence the reliability of compilers is critical, especially in the era of digitalization. To this end, automated compiler testing is an active research area which aims to automatically find bugs in production compilers. Various language-specific methodologies [Yang *et al.*, 2011; Livinskii *et al.*, 2020] have been proposed to generate random programs to test whether a compiler can correctly compile these programs; if the compiler crashes or hangs (*i.e.*, the compilation process does not terminate normally), or the compiled binary behaves differently from the source code, then a compiler bug is found. For example, Csmith [Yang *et al.*, 2011] is designed to randomly generate well-defined C programs[1] and can generate only C programs. Csmith has helped find hundreds of bugs in GCC and LLVM, and therefore has been integrated into the daily testing routine of GCC. However, it is non-trivial, time-consuming, and labor-intensive to design and implement such a *language-specific* program generator (referred to as LSG), which requires comprehensive language-specific domain knowledge to design correct, subtle program generation rules.

To minimize the cost of engineering a random program generator for compiler testing, researchers recently resorted to deep learning techniques [Liu *et al.*, 2019; Cummins *et al.*, 2018]. Such a deep learning-based program generator (referred to as DLG) attempts to automatically learn the syntactic and semantic rules of a programming language from human-written programs using a sophisticated deep learning model. Later, the DLG uses the trained model to generate programs to test the compiler of this language. Creating a DLG does not involve human efforts to encode domain knowledge into program generation rules, and this approach is demonstrated to be effective in compiler testing compared to LSGs. For example, DeepFuzz [Liu *et al.*, 2019] claimed that it triggers more code paths in GCC than Csmith.

However, we argue that it is unfair to use LSGs as the baseline to evaluate DLGs. An LSG usually has complex generation rules to ensure the generated programs are compilable and well-defined, in order to validate whether compilers

[1]A program is well-defined if the program does not contain any undefined behaviors; and an undefined behavior is a behavior that is not defined in the C language standard [IOS, 2005].

correctly optimize and produce binary code. These generation rules inevitably restrict the diversity of language features used in the generated test programs. For example, Csmith-generated programs only cover a small subset of the C language features, and mainly exercise the optimization algorithms in compilers but not the front end of compilers that handles lexing and parsing. By contrast, existing DLGs cannot warrant well-defineness or even compilableness, and may use arbitrary language features depending on the programs in the dataset. Therefore, it is not surprising to see that LSG-generated programs trigger lower code coverage in compilers than DLG-generated programs, especially code coverage in the front end of compilers; in terms of bug detection ability, it is also expected that LSGs such as Csmith do not trigger more bugs than DLGs because LSGs have been heavily used by various compiler communities in the past (*e.g.*, Csmith has been used for years on a daily basis) and their bug detection ability has saturated. Overall, using LSGs as baselines to evaluate DLGs likely leads to biased conclusions.

Kitten. To help researchers fairly evaluate DLGs with proper baselines, this paper proposes Kitten, a simple yet strong, fair baseline. Kitten is a *language-agnostic* program generation technique that supports abundant language features. Same as a DLG, Kitten requires a dataset consisting of programs. However, instead of the time-consuming step of training a deep neural model to create a DLG, Kitten directly generates new test programs by mutating the programs in the dataset. Specifically, given a program p in the dataset, Kitten randomly mutates the tokens of p or nodes in the parse tree [Aho *et al.*, 2006] of p, and outputs the mutation result as a new test program.[2] Kitten has obvious advantages over both LSGs and DLGs. Kitten is much easier to implement than LSGs as it does not need domain knowledge to carefully design the rules to generate programs; Kitten does not have the constraints defined by Csmith or similar work [Livinskii *et al.*, 2020], and thus supports diverse language features to comprehensively test compilers. Compared to DLGs, Kitten does not need to train a deep learning model, thus saving significant time and computational resources for training and generation; moreover, Kitten is interpretable, and extensible to support new mutation strategies.

Re-evaluating DLGs Using Kitten. Using Kitten as the baseline, we conducted a comprehensive experiment to empirically revisit the performance of two representative DLGs, *i.e.*, DeepSmith [Cummins *et al.*, 2018] and DeepFuzz [Liu *et al.*, 2019]. Considering the simplicity of Kitten, it is reasonable to expect that DLGs should at least perform similarly to Kitten. However, with 1,500-CPU/GPU-hour experiment and analysis, the results show that the performance of existing DLGs is far from this simple, reasonable objective in three perspectives: bug detection ability, the diversity of language features in the generated programs and code coverage of compilers. In 72-hour testing on GCC, DeepSmith triggers 3 hang bugs and 1 distinct crash, while Kitten triggers 1,750 hang bugs and 34 distinct crashes. The generated programs by Kitten cover 21,053 more lines and 26,853 more branches than the dataset, which is at least 2x as the one generated by DeepFuzz and DeepSmith. Moreover, the numbers of features leveraged in the generation by DeepSmith and DeepFuzz are also only around 64% of the one by Kitten. We believe that DLGs still have much room for improvement to compete against the simple baseline Kitten.

Contributions. We make the following contributions.

1. We identify that the evaluations of the state-of-the-art DLGs are biased due to the use of improper baselines.
2. We propose Kitten, a simple yet strong language-agnostic program generator as a fair baseline for evaluating DLGs.
3. We empirically demonstrate that DLGs have much room for improvement as they fail to compete with Kitten.
4. We make Kitten publicly available at https://doi.org/10.5281/zenodo.7946825 to benefit future research on DLGs.

2 Preliminary

2.1 Compilers and Compiler Bugs

Given a program p, a compiler translates the source code of p into binary code, so that p can be executed by a computer. A typical compilation process consists of three stages. First, the front end of the compiler parses the source code and builds an intermediate representation of p. Second, the middle end performs various optimizations like dead code elimination on the intermediate representation to make p run faster and use fewer resources. Lastly, the back end converts the optimized internal representation into binary code. Bugs can occur in any of the aforementioned stages in a compiler [Yang *et al.*, 2011; Sun *et al.*, 2016b; Livinskii *et al.*, 2020; Le *et al.*, 2015]. There are three major types of compiler bugs:

Crash. A compiler crashes, if it aborts the compilation process due to an error inside the compiler when compiling a program. For example, if a segmentation fault occurs when GCC is compiling a program, it aborts with an error message `internal compiler error: Segmentation fault`.

Hang. A hang (timeout) is a compiler bug when the compiler runs indefinitely to compile a program.

Miscompilation. Given a well-defined program p, a compiler miscompiles p, if the compiled binary of p is not semantically equivalent to p and thus behaves differently from p. This type of compiler bugs is referred to as miscompilation.

2.2 Program Generation for Compiler Testing

Automated compiler testing uses a program generator to automatically generate random test programs, and checks whether the compiler under test can correctly compile the generated programs. In this paper, we categorize program generators into the following two classes.

LSG: Language-Specific Program Generator

LSGs generate programs by following a set of language-specific rules that are meticulously crafted by human experts based on the domain knowledge of the language under test. Csmith [Yang *et al.*, 2011] is one representative LSG. It generates well-defined C programs that conform to

[2]Parsing a program into a list of tokens or a parse tree can be easily done with a parser generator such as Antlr [Antlr, 2022a], and the grammars of most main-stream programming languages are also available online [Antlr, 2022b].

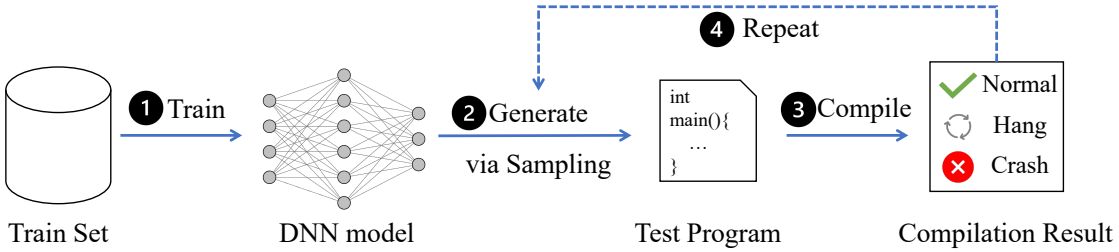


Figure 1: Workflow of using a DLG for testing a compiler.

the C language specification [IOS, 2005], in a top-down manner from a translation unit, functions and statements down to expressions. Each time a language construct is being generated, Csmith applies the generation rules to ensure that the generated program is compilable and well-defined. Several following studies extend this idea to other languages, such as OpenCL [Lidbury *et al.*, 2015] and CUDA [Jiang *et al.*, 2020].

Despite the success of LSGs in finding compiler bugs, one major limitation of LSGs is that engineering a LSG requires language-specific knowledge to design generation rules and strategies. For example, to adapt the idea of Csmith to a new language, developers need to comprehend each feature of the new language and engineer the corresponding generation rules to ensure that the generated programs are compilable. Considering the complexity of programming languages, such work is labor-intensive and time-consuming, and the generation rules cannot be easily generalized to other languages. Further, it is challenging to take into consideration all language features and their possible combinations when designing generation rules. In fact, most LSGs only support a subset of language features and thus the expressiveness of the generated programs is rather limited.

DLG: Deep Learning-Based Program Generator

To overcome the limitation of LSGs, researchers proposed to use deep learning techniques to automatically learn a program generator from existing programs. Figure 1 shows the overall workflow of learning a DLG and applying the DLG to find compiler bugs. From a given dataset consisting of human-written programs, ① a deep learning model is trained as a DLG, which automatically learns the syntactical and semantic features of the language. Specifically, the model encodes the probability of the next token given a sequence of tokens as a prefix. ② The trained model generates a test program by iteratively querying the model to compute the next token. Specifically, starting from a prefix like `int main`, the DLG samples the subsequent token from the probabilities encoded in the trained model, until some termination tokens are generated. Then ③ the generated test program is fed to compilers under test and the compilation result is checked to see if any crash or hang bug is triggered. ④ This process is repeated until the time limit is reached or a sufficient number of test programs are generated. DeepSmith [Cummins *et al.*, 2018] and DeepFuzz [Liu *et al.*, 2019] are the representative DLGs, which utilize Long Short-Term Memory (LSTM) model for training and generation. DSmith [Xu *et al.*, 2020b] and TSmith [Xu *et al.*, 2020a] also have similar workflows.

Property	DLGs	LSGs	Kitten
Need a set of programs for generation	✓	✗	✓
Language-agnostic	✓	✗	✓
Use arbitrary language features	✓	✗	✓
Warrant well-defineness/compilableness	✗	✓	✗
Can detect crash and hang bugs	✓	✓	✓
Can detect miscompilation bugs	✗	✓	✗

Table 1: Comparison of DLGs, LSGs, and Kitten.

Different from LSGs, DLGs can only be used to find compiler crashes and hang bugs, but not miscompilation bugs. This is because the DLG-generated programs are not guaranteed to be compilable or free of undefined behaviors, and it is difficult to automatically determine whether the generated programs are free of undefined behaviors [Yang *et al.*, 2011]. Table 1 summarizes the differences between DLGs and LSGs.

3 Revisiting the Evaluation of DLGs

As aforementioned, it is unfair to evaluate DLGs using LSGs and the resulting conclusions are likely to be biased. Thus, we aim to re-evaluate the performance of DLGs using a fair baseline. This section describes the design of our reevaluation experiment, and Kitten, a new fair baseline for evaluating DLGs.

3.1 Evaluation Methodology

To evaluate the performance of a program generator for compiler testing, we invoke it to continuously generate random programs for a certain period. After the generation, we feed them into a compiler under test for compilation. Finally, we measure the performance of this program generator from three perspectives: compiler bug detection ability, diversity of the language features in the generated programs, and code coverage. Each perspective and its metrics are introduced later in the corresponding sections. Since some measurements may affect the compiler's efficiency, all the measurements are conducted after the program generation is finished. For example, measuring code coverage requires instrumenting compilers, resulting in a longer compilation time of the generated programs.

3.2 Selected Program Generators

We selected two representative DLGs from literature—DeepFuzz and DeepSmith—and re-evaluated their perfor-

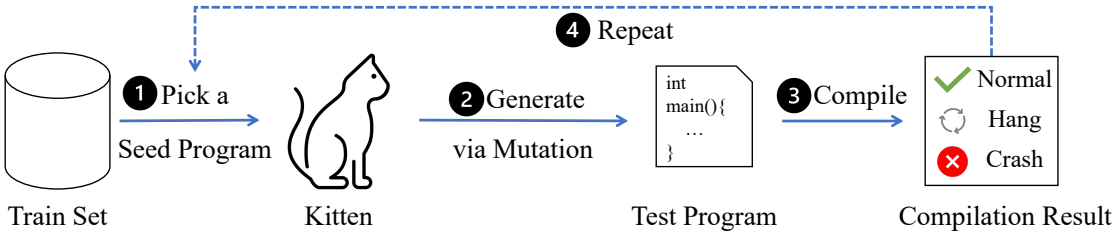


Figure 2: Workflow of using Kitten for testing a compiler.

mance. They are selected for two reasons. First, they were published in top-tier conferences of both software testing and artificial intelligence, representing the state of the art of DLGs. Second, their implementations are publicly available, avoiding re-implementation and thus minimizing threats to validity. Other work [Xu *et al.*, 2020a; Xu *et al.*, 2020b] is not selected since their implementations are not publicly available. For each program generator, we followed their documentation and tried our best efforts to reproduce their results. However, we found that the performance of DeepFuzz is not comparable to the one mentioned in its publication. Similar issues are also raised by other developers in its repository [Developers, 2022]. Nevertheless, since DeepFuzz and DeepSmith have very similar workflow and architecture, we believe DeepSmith is a reasonable representative DLG work. We also include Csmith in our experiment, one of the most commonly used LSGs for C compilers.

To fairly evaluate the performance of DLGs, a proper baseline is necessary. Specifically, we are looking for a baseline that is similar to DLGs but much simpler than DLGs. Since such a baseline does not exist in the literature, we propose a new, fair and simple baseline, Kitten. Kitten is similar to DLGs except not using a DNN. Considering its simplicity, we expect that DLGs should at least perform similarly to Kitten.

3.3 Kitten: A New, Fair and Simple Baseline

Overview. Figure 2 shows the program generation workflow of Kitten. Unlike LSGs, Kitten shares many similarities with DLGs. Kitten takes as input a dataset of programs and outputs a new set of programs iteratively. For each iteration, it ① randomly takes one seed program and ② applies a random mutation operation to the seed to create a new program. After the program generation, ③ these programs are fed into a compiler under test. If a crash or hang bug is triggered, a potential bug for this compiler is detected. ④ The above process is repeated until the time limit is reached or a sufficient number of test programs are generated.

The entire program generation process of Kitten is language-agnostic, especially the mutation operators. Kitten supports diverse mutation operators and it is easy to integrate others. We implemented two types of operators from literature [Aschermann *et al.*, 2019; Wang *et al.*, 2019], tree-level and token-level mutations.

Comparison with DLGs. Given a dataset of programs, DLGs leverage deep learning models to learn the syntactic and semantic rules and generate new programs, while Kitten generates random programs by directly mutating the programs in the dataset. Other than this, DLGs and Kitten share many commonalities as shown in Table 1: 1) generate new programs from a set of programs; 2) language-agnostic; 3) may use arbitrary language features ; 4) do not warrant well-defineness or even compilableness of the generated programs; 5) focus on detecting crash and timeout bugs in compilers.

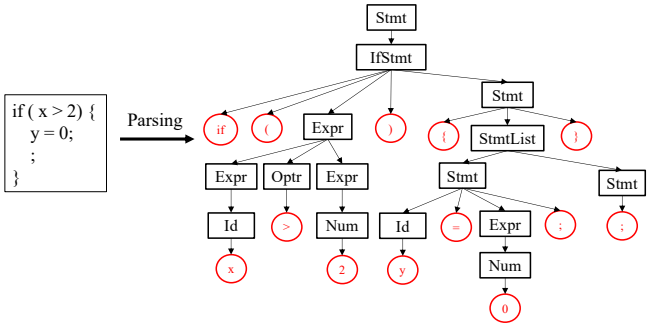


Figure 3: An example of a parse tree.

Tree-Level Mutation

Parse Tree. A parse tree is a tree representation of program's syntactical structure. Figure 3 shows a program and its parse tree. There are two types of nodes in a parse tree, *i.e.*, non-leaf nodes (black rectangles in Figure 3) and leaf node (red circles in Figure 3). The former one refers to a non-terminal symbol in language grammar, *e.g.*, expression and statement. The latter one refers to a terminal symbol that cannot have child nodes [Bison, 2022]. Parsing programs into parse trees can be easily done with a parser generator (like Antlr [Antlr, 2022a]) and corresponding grammar [Antlr, 2022b], both of which are generally available for common programming languages.

A tree-level mutation operator parses a seed program p to a parse tree t and mutates a sub-tree of t to generate a new program. The operator carefully checks the type of tree nodes and ensures the syntactic validity of the generated programs. In Kitten, we implemented three tree-level mutation operators [Aschermann *et al.*, 2019; Wang *et al.*, 2019] and they are illustrated in Figure 4. Intuitively, these mutations randomly replace one sub-tree st of the parse tree t with a new sub-tree st', under the constraint that the root nodes of st' and st represent the same non-terminal symbol, such as an expression and statement, The major difference among these mutations is the source of the new subtree st'. In *Sub-tree Replacement*,

	Program
Seed	int main(){int a = 1 * 2;}
Insertion	int main(){int a = 1 * 2; int}
Deletion	int main(){a = 1 * 2;}
Replacement	int main(){float a = 1 * 2;}

Table 2: Examples of token-level mutations.

Generator	Average	Standard Deviation
DeepFuzz	138.54	9.78
DeepSmith	315,919.06	1,947.83
Csmith	2,021.32	187.72
Kitten	3,001,079.25	548,262.38

Table 3: The number of programs generated per hour.

st' is randomly generated by Kitten using the language grammar. *Sub-tree Splicing* randomly selects a sub-tree from other programs in the dataset. *Recursive Sub-tree Repeat* attempts to find a st' in the ancestor of st, *i.e.*, st is a sub-tree of st', and then uses st' to replace st arbitrary times.

Token-Level Mutation

Token-level mutation operations tokenize a seed program into a list of tokens and directly mutate this list. We implemented three types of token-level mutation in Kitten, namely *insertion*, *deletion* and *replacement*. As their name implies, these mutations randomly insert, delete or replace a token in the list. Table 2 shows the examples of the three mutations. Different from tree-level mutations, token-level mutations do not guarantee that the produced mutant is syntactically correct. This is intended since syntactically incorrect programs can find bugs in the front end of compilers.

3.4 Miscellaneous

Compiler and Dataset. We use all program generators to test GCC, one of the most commonly used and tested C compilers. Following existing DLGs [Cummins *et al.*, 2018; Liu *et al.*, 2019], we constructed a dataset using all the C files of the testsuite of GCC 11.2. Kitten directly used this dataset to generate new programs, while DeepSmith and DeepFuzz are trained using this dataset until the loss is saturated.

Platform and Duration. To ensure each generator has the same computation resources, each generator is deployed on a unique GPU virtual machine on a cloud platform with the same configuration. The longest duration used by prior work is 48 hours in DeepSmith. We choose a longer duration, *i.e.* 72 hours, for a comprehensive evaluation.

Noted that the program generation efficiency is not the primary concern in compiler testing, while effectiveness is more important. Nevertheless, we show the number of programs generated per hour in Table 3 for reference. DeepSmith and DeepFuzz have significantly different generation speeds. Such difference may be due to the aforementioned implementation issues in DeepFuzz. Csmith takes more effort than DLG and Kitten to ensure the semantic correctness of generated programs, resulting in a lower generation speed. Kitten is a better baseline than Csmith for evaluating DLGs since DLGs even do not guarantee compilableness of generated programs. Kitten has a much higher generation speed (9.5x at least) than DLGs since it directly mutates the parse tree or toke list, which requires less computation resource.

4 Empirical Findings

This section presents the empirical findings and discusses their implications.

4.1 Bug Detection Ability

Bug detection ability is one of the most important metrics to measure the quality of generated programs. In this RQ, we measured the number of bugs triggered by the program generated by each generator. We fed all the programs into the latest GCC development version (commit id gf7a3ab) and measured the number of crash and hang bugs. Specifically, for crash bugs, we first leverage program reduction tool Perses [Sun *et al.*, 2018] to reduce the bug-triggering program to the smallest amount of code that still replicates the bug. Second, we analyze the stack traces and error messages of bugs. For bugs that have similar stack traces and error messages, we cluster them into one group and then investigated them manually. For hang bugs, as compilers do not throw error messages when hanging, there is no automatic way to deduplicate them and thus we only counted the total numbers. We did not measure the miscompilation bugs since it is a research challenge to automatically detect such bugs [Le *et al.*, 2014; Livinskii *et al.*, 2020] and neither DLGs nor Kitten is designed to find miscompilations.

Distinct Crash. Figure 5 shows the cumulative number of distinct compiler crashes triggered by Kitten and DeepSmith. In total, the programs generated by DeepSmith triggered 181 crash bugs but they all have the same root cause. In other words, DeepSmith only found one distinct crash bug. The programs generated by Kitten trigger 2,354 crashes and 34 of them are distinct bugs, which significantly outperforms DeepSmith. DeepFuzz and Csmith did not find any crash bugs.

Hang. Following the practice in compiler testing [Yang *et al.*, 2011; Le *et al.*, 2014], we used 120 seconds as the timeout threshold. DeepSmith triggered 3 hang bugs in GCC in 72 hours while Kitten triggered 1,750 hang bugs. DeepFuzz and Csmith do not trigger any hang bug.

We reduced the bug-triggering programs using Perses [Sun *et al.*, 2018] and reported the discovered bugs to GCC after excluding the bugs that have already been reported recently. Figure 6 shows four of the bugs found by Kitten. All four bugs are new, and confirmed by GCC developers. Bug 105555 has been present since at least GCC 4.8.0, and was not found by any testing technique for over nine years. Bug 105554 has been fixed by developers. Other bugs found by Kitten are awaiting further analysis from developers. Given that Kitten only takes three days to find four confirmed bugs, Kitten is an effective baseline in terms of bug detection.

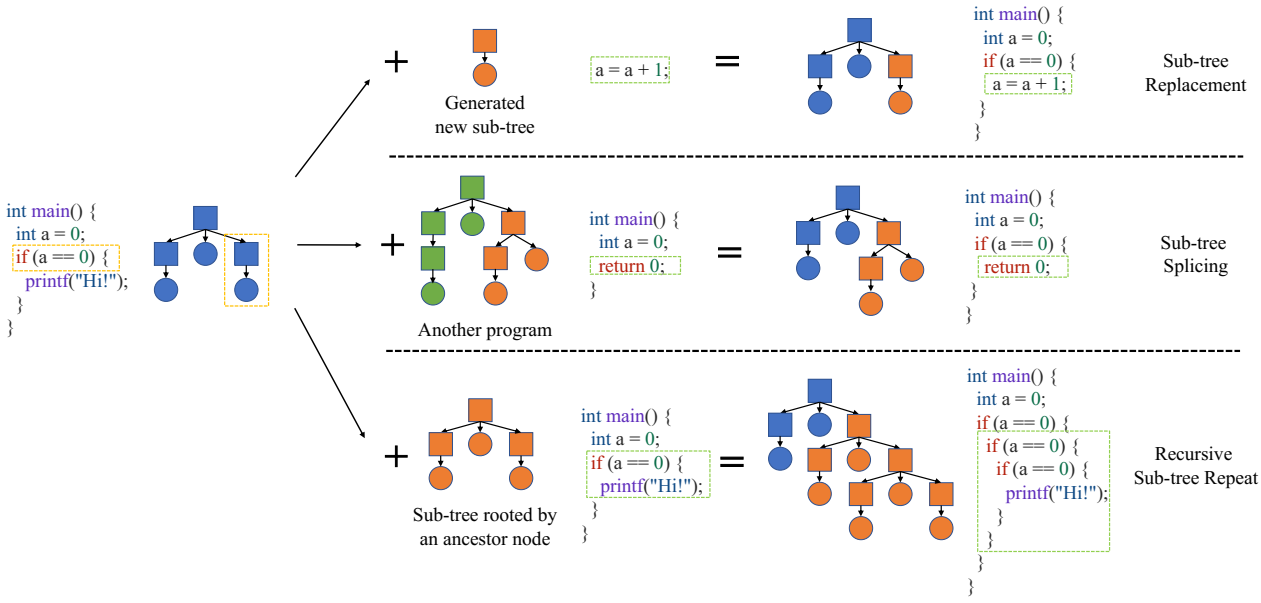


Figure 4: Tree-level mutations. The parse trees are simplified for illustration, thus not exactly matching the code examples.

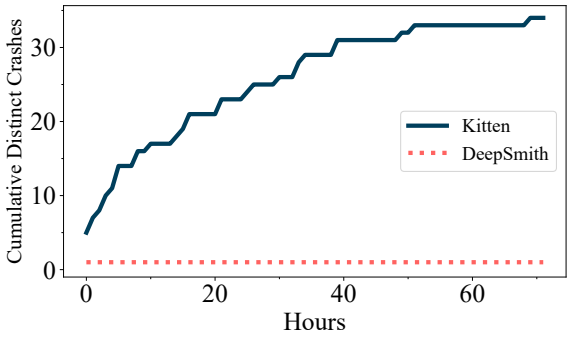


Figure 5: Distinct crashes. DeepFuzz and Csmith are not shown in this figure since they do not trigger any crash bugs.

```
void foo() { __asm__("" :: "m"(({if(8);})));}
```

(a) GCC-100501, Crash

```
static a(); b(void) {sizeof ( int [ a}
static c(); d(void) {sizeof((int[c
```

(b) GCC-104764, Hang

```
__attribute__((target_clones(
"arch=core-avx2", "default")))
a(__attribute__((__vector_size__(
4 * sizeof(long)))) long) {}
```

(c) GCC-105554, Crash

```
a() { &__imag *(_Complex *)a
```

(d) GCC-105555, Crash

Figure 6: Bugs found by Kitten, including their bug-triggering programs, bug ids, and symptoms.

Compared to Csmith, it seems that DLGs have a good bug detection ability since they find more bugs than Csmith. However, such an advantage is due to the unfair comparison since Csmith has been integrated into the daily testing of GCC and any triggered issues are expected to be fixed in development already [Livinskii *et al.*, 2020]. In contrast, the limitation of DLGs in bug detection is demonstrated when Kitten is the evaluation baseline: the number of bugs detected by DLGs is far less than the one of Kitten, despite Kitten is a simple baseline without using complicated DNN models.

Finding and Advice 1: The bug detection abilities of DLGs are limited and did not outperform Kitten, a simple baseline without using DNN models. Future research should improve the bug detection abilities of DLGs to outperform Kitten.

4.2 Diversity of Language Features

To comprehensively test compilers, the generated program should cover a diverse set of language features. Following an existing study [Dong *et al.*, 2022], we use the number of AST node types as a proxy metric of language features. Abstract Syntax Tree (AST) is a tree representation produced by compilers in the compilation. Each AST node type is regarded by compilers as a distinct type of component defined in language grammar, such as the variable declaration, function call, *etc.* For each generator, we sampled all the programs generated at first, twenty-fifth and forty-ninth hours and counted the number of distinct AST node types.

Figure 7 presents the result in a Venn diagram. The programs generated by DeepFuzz and DeepSmith contain 91

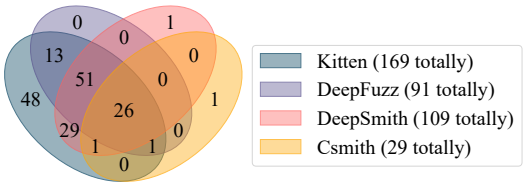


Figure 7: AST node types

and 109 types of AST nodes, respectively. Both outperform Csmith which only utilizes 29 node types. This is because the generation rules embedded in Csmith only use a subset of language features to ensure the semantic correctness of the generated programs. Including additional features requires the developers manually design complex rules to guarantee correctness, which is a challenging task. As a result, it is unfair to compare DLGs with Csmith since DLGs do not explicitly care about such correctness of programs.

Kitten is a much fairer baseline for DLGs than Csmith, since neither DLGs nor Kitten limits the features they can use in program generation. The programs generated by Kitten have the largest number of distinct AST node types, *i.e.* 169. Moreover, all the AST node types in DeepFuzz and 108 out of 109 types in DeepSmith are included by the programs generated by Kitten. Based on this result, we may conclude that DLGs do not fully learn and leverage the language features in the training set. A possible reason is that, when sampling the next token, DLGs prefer the token that has a high likelihood according to co-occurrence. However, DLGs have limited knowledge of the syntactic and semantic meaning of tokens and fail to generate the programs toward diversity.

Finding and Advice 2: Existing DLGs do not fully utilize the language features in program generation, which may affect their bug detection abilities. Future research should encourage DLGs to use more diverse language features.

4.3 Code Coverage

Code coverage is an important metric in compiler testing [Livinskii *et al.*, 2020; Sun *et al.*, 2016a; Le *et al.*, 2015]. It measures the code in compilers that is executed when compiling programs. High code coverage typically indicates that diverse code logic paths in compilers are exercised. For each set of generated programs, we fed them into GCC and used LCOV to measure the code coverage using three commonly used metrics: line, branch and function coverage.

Figures 8a to 8c shows the number of lines, branches and functions covered by the programs generated by each generator but not by the seed programs. DeepSmith covers 2,175 lines, 2,562 branches, and 17 functions that are not covered by the seed programs, while Csmith covers 9,650 lines, 12,510 branches, and 206 functions. Even though DeepSmith generated much more (156x) programs and leveraged more (109 vs 29) features than Csmith, the code coverage achieved by DeepSmith is much lower than Csmith.

To understand the reason, we carefully investigated the difference between their coverage. Specifically, we analyzed how many new covered branches are in the front end of GCC and how many of them are in the middle/back end. Figure 9 shows the results. It is clear that the programs generated by DeepSmith mainly cover new branches in the front end of compilers, such as lexer and parser. By contrast, most of the branches covered by programs generated by Csmith are located in the middle/back end of compilers. Although the programs generated by Csmith only include limited language features, they can trigger complex optimization in the middle/back end of compilers, resulting in high code coverage. This result also demonstrated the effectiveness of language-specific rules designed by experts in compiler testing.

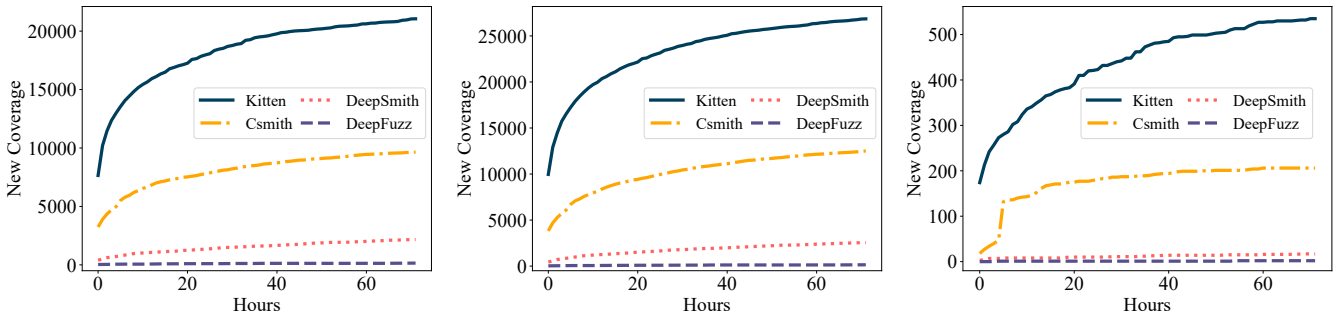
The line, branch, and function coverage achieved by Kitten is 9.68x, 10.48x, and 31.47x as DeepSmith, respectively. As shown in Figure 9, such improvements locate in both the front end and middle/back end of compilers. Note that the advantage over DeepSmith is not only because of program generation speed. Furthermore, the programs generated by DeepSmith after seventy-two hours achieved lower coverage than the programs generated by Kitten in the first hour, despite DeepSmith generating more programs in those seventy-two hours than Kitten in the first hour.

Finding and Advice 3: DLGs does not outperform LSGs and Kitten in terms of code coverage. One of the reasons is that the programs generated by DLGs do not trigger diverse optimizations in compilers, which can be a promising research direction.

4.4 Implications

Our experiment results show that DLGs like DeepSmith do not achieve outstanding performance in compiler testing. They have certain advantages over Csmith but such advantages are due to the unfair comparison mentioned in §2.2. As we mentioned earlier, Csmith is designed to leverage a limited set of language features to generate semantically correct programs, it is natural that the diversity of generated programs is not as good as DLGs. Our experiment also shows that once the evaluation baseline is switched to Kitten, the performance of DLGs is worse than Kitten in all three perspectives: bug detection ability, diversity of language features, and code coverage. Although DLGs attempt to leverage DNN models to learn the information of programs from datasets and generate new programs for compiler testing, it turns out that the results are not as good as the approach of Kitten, *i.e.*, directly mutating the programs in datasets.

We believe Kitten can benefit future DLG-related research in various aspects. First, Kitten sets a new fair baseline for DLGs. Future work of DLGs should at least have a significant improvement over Kitten. Second, the fact that Kitten outperforms existing DLGs may enlighten future research. Their limited performance implies that a simple end-to-end deep learning approach without incorporating any domain-specific information has not achieved decent performance in compiler testing. A promising research direction is to incorporate domain-specific knowledge extracted from the pro-



(a) Line Coverage (b) Branch Coverage (c) Function Coverage

Figure 8: Code coverage of the programs generated by Kitten, Csmith, DeepSmith and DeepFuzz.

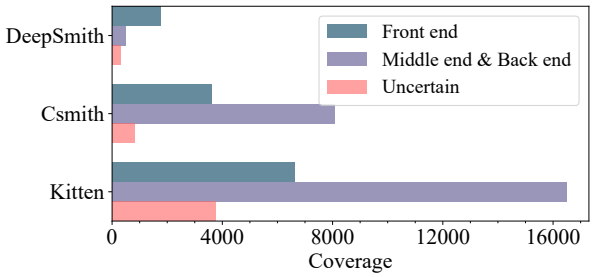


Figure 9: Distributions of new branch coverage. Uncertain refers to the cases of which category cannot be determined.

grams, such as parse trees, control-flow graphs and data-flow graphs, into DNN models. Moreover, as Kitten itself is shown to be effective, future research may study how to improve the workflow of Kitten using machine learning techniques.

4.5 Other Discussion

Please note that we choose to control the time and computation resources since it is a common practice in software testing to evaluate testing techniques by setting a limit on time and computation resources [Zhang *et al.*, 2022; Klees *et al.*, 2018]. For example, Google FuzzBench deploys each fuzzer for 24 hours. We believe this practice is related to how the software is tested in the industry, *i.e.*, developers need to adequately test the software before shipment date using the limited computation resources available to them.

We did not control the number of generated programs since the number of programs is easy to be tampered with. For example, one may easily merge multiple programs outputted by Kitten into one program or vice versa. Moreover, even if DLGs, LSGs, and Kitten are evaluated using the same number of generated programs, the conclusion of our empirical study remains the same. For bug detection ability, Figure 5 demonstrates that the number of crash bugs found by Kitten in the first hour is more than the number of crash bugs found by DeepSmith in 72 hours, although the number of programs generated by Kitten in the first hour is smaller than the number of programs generated by DeepSmith in 72 hours. As for the diversity of language features, Kitten covers 168 AST node types in the first hour, while DeepSmith covers 109 AST node types in 72 hours. The results related to code coverage have been discussed in §4.3.

5 Related Work

Compiler testing has been actively explored for many years. A common approach is automatically generating test programs with a generator and checking whether compilers properly compile these programs. As mentioned in §2.2, LSGs and DLGs are two main kinds of generators. Kitten is much easier than LSG to implement as it does not require expert knowledge and massive engineering efforts to ensure the validity of generated programs. Different from DLGs that train DNN models using a given dataset, Kitten generates new programs by randomly mutating the ones in the dataset. Another mainstream of compiler testing is detecting miscompilation bugs. For example, EMI [Le *et al.*, 2014; Sun *et al.*, 2016a] generates a set of mutated programs that are equivalent to each other *w.r.t.* a set of inputs. After these programs are compiled, if the binary programs behave differently given these inputs, it indicates that at least one of them is miscompiled and a defect is detected. These approaches requires the insightful language-specific knowledge of developers to propose mutations that preserve the equivalence relationship *w.r.t.* a set of inputs, primarily targeting miscompilation bugs. In comparison, Kitten is a language-agnostic random program generator, targeting crash and hang bugs.

6 Conclusion

This study argues that the evaluations of the DLGs are biased due to the improperly chosen baselines. LSGs are designed to utilize limited language features to generate well-defined programs, while DLGs generate arbitrary programs without concerns about the syntactic and semantic correctness. We revisited the evaluation of DLGs using Kitten, a fair, simple and strong baseline for DLGs. Instead of using DNN models, Kitten directly derives new programs from the dataset. Empirical results show that the advantage of DLGs claimed in their publications is likely due to the biased selection of baseline. Despite the simplicity of Kitten, DLGs cannot compete with Kitten in multiple metrics. With in-depth analysis of the evaluation results, we discuss potential directions for advancing future research on DLGs, and strongly believe that Kitten is the fair, right baseline for evaluating DLGs.

Acknowledgments

This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant and CFI-JELF Project #40736. We express our gratitude to Microsoft for generously providing Azure credits for this research.

References

[Aho *et al.*, 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, chapter 2.2.3. ParseTree. Addison Wesley, August 2006.

[Antlr, 2022a] Antlr. Antlr. https://www.antlr.org/, 2022. Online; accessed 26-July-2022.

[Antlr, 2022b] Antlr. Antlr grammar. https://github.com/antlr/grammars-v4, 2022. Online; accessed 26-July-2022.

[Aschermann *et al.*, 2019] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[Bison, 2022] Bison. Bison grammar files. http://web.mit.edu/gnu/doc/html/bison_6.html#SEC34, 2022. Online; accessed 26-July-2022.

[Cummins *et al.*, 2018] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 95–105, New York, NY, USA, 2018. Association for Computing Machinery.

[Developers, 2022] Developers. Github issue for deepfuzz. https://github.com/s3team/DeepFuzz/issues/1, 2022. Online; accessed 26-July-2022.

[Dong *et al.*, 2022] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. Bash in the wild: Language usage, code smells, and bugs. *ACM Trans. Softw. Eng. Methodol.*, apr 2022. Just Accepted.

[IOS, 2005] IOS. ISO/IEC 9899:TC2: Programming Languages—C. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, 2005.

[Jiang *et al.*, 2020] Bo Jiang, Xiaoyan Wang, W. K. Chan, T. H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. Cudasmith: A fuzzer for cuda compilers. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 861–871, 2020.

[Klees *et al.*, 2018] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[Le *et al.*, 2014] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014.

[Le *et al.*, 2015] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In Jonathan Aldrich and Patrick Eugster, editors, *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*, pages 386–399. ACM, 2015.

[Lidbury *et al.*, 2015] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. *SIGPLAN Not.*, 50(6):65–76, jun 2015.

[Liu *et al.*, 2019] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1044–1051, Jul. 2019.

[Livinskii *et al.*, 2020] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[Sun *et al.*, 2016a] Chengnian Sun, Vu Le, and Zhendong Su. 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, page 849–863, New York, NY, USA, 2016. Association for Computing Machinery.

[Sun *et al.*, 2016b] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery.

[Sun *et al.*, 2018] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, page 361–371. Association for Computing Machinery, 2018.

[Wang *et al.*, 2019] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[Xu *et al.*, 2020a] Haoran Xu, Shuhui Fan, Yongjun Wang, Zhijian Huang, Hongzuo Xu, and Peidai Xie. Tree2tree structural language modeling for compiler fuzzing. In Meikang Qiu, editor, *Algorithms and Architectures for Parallel Processing*, pages 563–578, Cham, 2020. Springer International Publishing.

[Xu *et al.*, 2020b] Haoran Xu, Yongjun Wang, Shuhui Fan, Peidai Xie, and Aizhi Liu. Dsmith: Compiler fuzzing through generative deep learning model with attention. In

2020 International Joint Conference on Neural Networks (IJCNN), pages 1–9, 2020.

[Yang *et al.*, 2011] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[Zhang *et al.*, 2022] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3699–3715, Boston, MA, August 2022. USENIX Association.