DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing

Xiao Liu, Xiaoting Li, Rupesh Prajapati, Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA

Abstract

Compilers are among the most fundamental programming tools for building software. However, production compilers remain buggy. Fuzz testing is often leveraged with newly-generated, or mutated inputs in order to find new bugs or security vulnerabilities. In this paper, we propose a grammar-based fuzzing tool called DEEPFUZZ. Based on a generative *Sequence-to-Sequence* model, DEEPFUZZ automatically and continuously generates well-formed C programs. We use this set of new C programs to fuzz off-the-shelf C compilers, e.g., GCC and Clang/LLVM. We present a detailed case study to analyze the success rate and coverage improvement of the generated C programs for fuzz testing. We analyze the performance of DEEPFUZZ with three types of sampling methods as well as three types of generation strategies. Consequently, DEEPFUZZ improved the testing efficacy in regards to the line, function, and branch coverage. In our preliminary study, we found and reported 8 bugs of GCC, all of which are actively being addressed by developers.

Introduction

Compilers are among the most important software of computing systems and they are typically part of the trust computing base, but they remain buggy. For example, GCC, a long-lasting software released in 1987, is a standard compiler for many Unix-like operating systems. Over 3,410 internal bugs (Yang et al. 2011) have been caught since it is created. Similarly, for Java, Python, and JavaScript, thousands of bugs have been found in those widely-used compilers and interpreters. These compiler bugs can result in unintended program executions and lead to catastrophic consequences in security-sensitive applications. It may also hamper developer productivity in debugging a program when the root cause cannot be decided in the applications or compilers. Therefore, it is critical to improve the compiler correctness. But it is not easy to validate compilers with the growing code base: the code base of today's GCC is around 15 million lines of code (Sun et al. 2016), close to the entire Linux kernel, which is around 19 million lines of code.

It is critical to make compilers dependable. In the past decade, compiler verification has been an important and active area for the verification grant challenge in computing research (Hoare 2003). Mainstream research focuses on formal verification (Leroy and Grall 2009), translation validation (Necula 2000), and random testing (Lidbury et al. 2015; Le, Afshari, and Su 2014; Le, Sun, and Su 2015). The first two categories try to provide certified compilers. For example, CompCert (Leroy et al. 2016) has made promising progress in this area. But in practice, it is challenging to apply formal techniques to fully verify a production compiler such as GCC, especially when the proof is not constructed together with the compiler. Therefore, testing remains the dominant approach in compiler validation.

Our work focuses on compiler testing. By feeding in programs covering different features to different production compilers turning on different levels of optimizations, internal compiler errors (genuine bugs of the compiler) may be triggered during the compilation with a detailed error message indicating where and what the error is. However, it is challenging to generate "good" programs to make testing more efficient and to build a continuous testing framework by automating this process. Each test, including man-crafted ones, in the existing methods, covers some features and it is common today to see larger and larger test suites for modern compilers. This improves the testing coverage but it takes a lot of human effort to construct these tests. Nevertheless, a practical way to reduce human labor for testing is fuzz testing, or fuzzing.

Fuzzing (Bird and Munoz 1983) is a method to find bugs or security vulnerabilities. A program is repeatedly executing with automatically generated or modified inputs to detect abnormal behaviors such as program crashes. Main techniques for input fuzzing in use today are black box random fuzzing (Zalewski 2015), white box constraint-based fuzzing (Godefroid, Kiezun, and Levin 2008), and grammar-based fuzzing (Dewey, Roesch, and Hardekopf 2014). Black box and white box fuzzing are fully automatic, and have historically been proven to be effective in finding security vulnerabilities in binary-format file parsers. In contrast, grammar-based fuzzing requires an input grammar specifying the input format of the application under test, which is typically written by hand. This process is laborious, time-consuming, and error-prone. However, grammar-based fuzzing is the most effective fuzzing technique known today for fuzzing applications with complexly structured input formats, e.g., compilers. In the scenario of compiler testing, one way to deploy the grammar-based fuzzing is to encode the C grammar as rules for test case generation. But in practice, C11 (of the International Organization for Standardization (ISO) 2011), the current standard of the C programming language, has 696 pages of detailed specifications, which brings the hurdle for engineers to construct such a grammar-based engine.

In this paper, we consider the problem of automatically generating syntactically valid inputs for grammar-based fuzzing with a generative recurrent neural network. To be more specific, we aim to train a generative neural network to learn the "grammar", or to be more precise, the language patterns, of the input data. We propose to train a *Sequence-to-Sequence* model (Sutskever, Vinyals, and Le 2014) in a supervised learning strategy, leveraging the original test suites provided with production compilers. Originally, the *Sequence-to-Sequence* model is widely used for machine translation (Klein et al. 2017) and text generation (Sutskever, Martens, and Hinton 2011). Theoretically speaking, by training a model on the original paragraphs, we implicitly encode the correct spelling of words, valid syntaxes of sentences, detailed styles of writing behaviors into a generative model. The same idea can be applied to program synthesis, where we only need to train a model to generate different syntactically valid programs on top of a seed data set. For the training data set, we adopted the original GCC test suite where there are over 10,000 short, or small, programs that cover most of the features specified in the C11 standard. On the training stage, we tune parameters to encode the language patterns for C programs into the model, based on which, we continuously generate new programs for compiler fuzzing.

Contributions. Our work is the *first* to use a generative recurrent neural network for grammar-based compiler fuzzing.

- First, the proposed framework is fully automatic. By training a *Sequence-to-Sequence* model which can be viewed as an implicit representation of the language patterns for the training data, C syntax in our context, our framework DEEPFUZZ will continuously provide new syntactic correct C programs.
- Second, we build a practical tool for fuzzing off-the-shelf C compilers. We conduct a detailed analysis of how key factors will affect the accuracy of the generative model and fuzzing performance.
- Third, we apply our DEEPFUZZ technique to test GCC and Clang/LLVM. During our preliminary analysis, the testing coverage (line, function, and branch) is increased and we have found and reported 8 real-world bugs.

Overview

Sequence-to-Sequence Model

We build DEEPFUZZ on top of a *Sequence-to-Sequence* model, which implements two recurrent neural networks (*RNNs*) for character-level sequences prediction. An *RNN* is a neural network that consists of hidden states $\mathbf{h}$ and an optional output $\mathbf{y}$. It operates on a variable-length sequence, $\mathbf{x} = (x_1, x_2, ..., x_T)$. At each step t, the hidden state $h_{\langle t\rangle}$ of the RNN is updated by

$$h_{\langle t\rangle} = f(h_{\langle t-1\rangle}, x_t) \tag{1}$$

where f is a non-linear activation function. An RNN can learn a probability distribution over a sequence of characters to predict the next symbol. Therefore, at each timestep t, the output from the RNN is a conditional distribution $p(x_t|x_{t-1}, ..., x_1)$. For instance, in our case, upon a multinomial distribution of the next character, we use a softmax activation function for the output

$$p(x_{t,j} = 1|x_{t-1}, ..., x_1) = \frac{\exp(w_j h_{\langle t\rangle})}{\sum_{j=1}^{K} \exp(w_j h_{\langle t\rangle})}, \tag{2}$$

for all possible symbols $j = 1, ..., K$, where w_j are the rows of a weight matrix W. By combining these probabilities, we compute the probability of the sequence x using

$$p(x) = \prod_{t=1}^{T} p(x_t|x_{t-1}, ..., x_1). \tag{3}$$

With the learned distribution, it is straightforward to generate a new sequence by iteratively sampling new characters at each time step.

A *Sequence-to-Sequence* model consists of two RNNs, an *encoder* and a *decoder*. The *encoder* learns to encode a variable-length sequence into a fixed-length vector representation and the *decoder* will decode this fixed-length vector representation into a variable-length sequence. It was originally proposed by Cho et al. (2014) for statistical machine translation. The encoder RNN reads each character of an input sequence x while the hidden states of the RNN changes. After reading the end of this sequence, the hidden state of the RNN is a summary c of the whole input sequence. Meanwhile, the decoder RNN is trained to generate the output sequence by predicting the next character y_t given the hidden state $h_{\langle t\rangle}$. However, unlike a pure RNN, both y_t and $h_{\langle t\rangle}$ are also conditioned on y_{t-1} and the summary c of the input sequence. In this case, to compute the hidden states of the decoder, we have

$$h_{\langle t\rangle} = f(h_{\langle t-1\rangle}, y_{t-1}, c), \tag{4}$$

and similarly, the condition distribution of the next character is

$$p(y_t|y_{t-1}, ...y_1, c) = g(h_{\langle t\rangle}, y_{t-1}, c), \tag{5}$$

where f and g are activation functions. Overall, the two RNNs *Encoder-Decoder* are jointly trained to generate a target sequence given an input sequence.

All RNNs have feedback loops in the recurrent layer. This design allows them to maintain information in "memory" over time. However, it can be difficult to train standard RNNs to learn long-term temporal dependencies, but which are common in programs. This is because the gradient of the loss function decays exponentially with time (Chung et al. 2014). Therefore, in our design, we adopt a variant of RNN, long short-term memory (LSTM), specifically in our encoder and decoder. LSTM units include a "memory cell" that can keep information in memory for long periods of time, in which case long history information can be stored.

In previous studies, the *Sequence-to-Sequence* model has been trained to generate syntactically correct PDF objects to fuzz a PDF parser (Godefroid, Peleg, and Singh 2017). The core idea behind this work is that the source language syntax can be learned as a by-product of training on string pairs. Shi, Padhi, and Knight (2016) investigated with an experiment that the *Sequence-to-Sequence* model can learn both local and global syntactic information about source sentences. This work lays a foundation for formal language synthesis with RNN. In our paper, we apply a similar idea for compiler fuzzing. During the training, we split the sequence into multiple training sequences of a fixed size d. By cutting the sequences, we have the i^{th} training sequence $x_i = s[i*d : (i+1)*d]$, where $s[k : l]$ is the subsequence of s between indices k and l. The output sequence for each training sequence is the next character, i.e., $y_t = s[(i+1)*d+1]$. We configure this training process to learn a generative model over the set of training sequences.

Workflow

In general, we propose DEEPFUZZ for two main objectives. The first is to generate new programs that follow legitimate grammars from a set of syntactically correct programs. The major challenge comes from long sequence handling and language grammar representing. The second objective is to improve the compiler testing efficacy. We target at improving the coverage and capturing more internal errors in production compilers.[1]

Figure 1 shows the workflow of DEEPFUZZ. There are two stages in the entire workflow, *Program Generation* and *Compiler Testing*. We target on production compilers such as GCC, the GNU Compiler Collection (2018) and LLVM/-Clang (Clang: a C language family frontend for LLVM 2018). On the first stage, we train a generative *Sequence-to-Sequence* model with collected data from the original man-crafted compiler test suites. Before we feed the sequences into the training model, we preprocess them to avoid noise data. We detail the *preprocess* step later in *Preprocessing*. The model we are going to fit is a general *Sequence-to-Sequence* model that has 2 layers with 512 hidden units for each layer. We compare our model configuration with the state-of-the-art sequence generation studies in *Experiment Setup*. For program generation, we try different generation strategies. We detail the generation strategies and their rationale in *Generation Strategy*. Because our target is to fuzz production compilers, we aim at generating programs that cover the most features of the C language. Therefore, we also adopted some sampling methods as detailed in *Sampling Variants*, to diversify the generated programs.

[1]An internal compiler error, also abbreviated as ICE, is an error during the compilation not due to the erroneous source code, but rather results from bugs of the compiler itself (Cleve and Zeller 2005). Usually, it indicates inconsistencies being found by the compiler. Commonly, the compiler will output an error message like the following: *gcc: internal compiler error: Illegal instruction (program). Please submit a full bug report, with preprocessed source if appropriate.*

On the second stage, we feed the generated C programs, either syntactically correct or incorrect, to the compilers in different optimization levels and log the compiling messages. In addition to the compiling message, we log the execution trace to provide the coverage information. In summary, for this program generation task, we have three objectives: to generate syntax valid programs, to improve code coverages, and to detect new bugs. We perform studies on three related metrics, *pass rate*, *coverage*, and *bugs*, for the three objectives in *Evaluation*.

Design

We propose DEEPFUZZ to continuously generate syntactically correct C programs to fuzz production compilers. As described in *Overview*, the complete workflow contains two stages, *Program Generation* and *Compiler Testing*. In this section, we present more details.

Preprocessing

Before we set up the training stage, we split the sequence into multiple training sequences of a fixed size. The output sequence for each training sequence is the next character right next to an input sequence. We configure this training process to learn a generative model over the set of all training sequences. However, we notice that there is some noise in the concatenated sequence which needs to be well-handled. In preprocessing, we mainly take care of three issues: *comment*, *whitespace*, and *macro*.

Comment. We first remove all the comments, including line comments and block comments using patterns in regular expression from the training data.

Whitespace. According to the POSIX standard, whitespace characters include common space, horizontal tab, vertical tab, carriage return, newline, and feed. To unify program style, we replaced all the white space characters with a single space.

Macro. Macro is a common feature of the C programming language. A macro is a fragment of code which has been given a new name. In our implementation, whenever the name is used, it is always replaced by the contents of the macro.

Sampling Variants

We use the learnt *Sequence-to-Sequence* model to generate new C programs. With a prefix sequence "`int `", for example, it is highly possible for the learnt distribution to predict "`main`" to follow up. However, our target is to diversify original programs to have more generated statements like "`int foo = 1;`" or "`int foo = bar(1);`". Therefore, we propose to adopt some sampling methods to sample the learnt distribution. We describe the three sampling methods that we employ for generating new C programs here: *NoSample*, *Sample* and *SampleSpace*.

NoSample. In this sampling method, we directly rely on the learnt distribution to greedily predict the best next character given a prefix.

Sample. To overcome the limitation of the *NoSample* method, given a prefix sequence we propose to sample next character instead of picking the top predicted one.

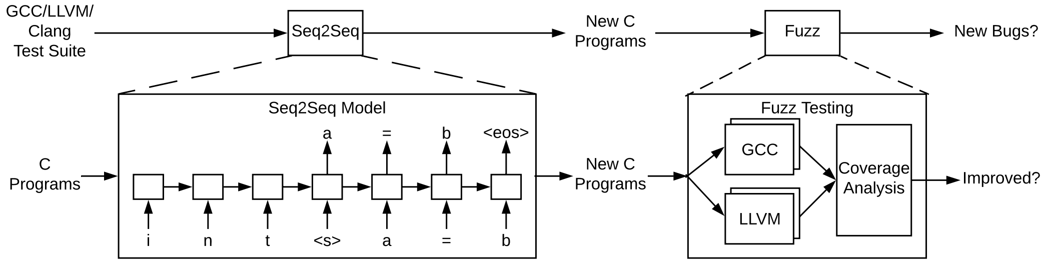


Figure 1: Workflow of DEEPFUZZ

SampleSpace This sampling method is a combination of *Sample* and *NoSample*. In this method, we only sample the next character among all the predicted ones over the threshold when the prefix sequence ends with a whitespace.

Generation Strategy

To continuously fuzz production compilers, we use the learnt model to generate new sequences of the C programming language. We treat programs in the original test suites as seeds. Based on a sequence from the original program as the prefix, we will generate new code. To make the most of the generated sequences, we propose three generation strategies: *G1)* we insert the newly generated code based on the same prefix sequence at one place into the original well-formed programs; *G2)* we generate new pieces of code, but they will be generated with prefix sequences randomly picked from different locations in the original program and, then insert back respectively; *G3* we chop out the same number of lines[2] after the prefix sequence from the original program and insert the newly generated new lines into the position of the sentences that have been chopped out. Moreover, more generation strategies can be conveniently set up within our framework but we perform a preliminary study with these three kinds.

Evaluation

Experiment Setup

To evaluate DEEPFUZZ, we pipelined a prototyping workflow which trained a *Sequence-to-Sequence* model based on a set of syntactically correct C programs. Originally, the training data set, which contains 10,000 well-formed C programs, was collected and sampled from the GCC test suites. We trained a *Sequence-to-Sequence* model with 2 layers and there are 512 LSTM units per layer. We set the dropout rate of 0.2. We have released the source code[3] for public dissemination.

In a previous study on text generation (Sutskever, Martens, and Hinton 2011), researchers trained a one-layer *RNN* with over 100 MB of training data, and there are 1,500 hidden units in this one-layer model. For the closest related work, Learn&Fuzz (Godefroid, Peleg, and Singh 2017), which adopted a generative *Sequence-to-Sequence* model to generate new PDF objects for PDF parser fuzzing, researchers trained a model with two layers and in each of these layers, there are 128 hidden units. They trained this model over a data set containing 534 well-formed PDF files. In our study, we trained a model with two layers where there are 512 LSTM units in each layer of the DEEPFUZZ framework. The training data set, which contains 10,000 syntactically correct C programs sampled from production compiler test suites, is larger than any previous studies.

We trained the *Sequence-to-Sequence* model in a supervised setting. In order to analyze the training performance, we trained multiple models parameterized by the number of passes, or *epochs*. An *epoch* is defined as an iteration of the learning algorithm to go over the complete set of training data. We trained the model for 50 epochs on a server machine with 2.90GHz Intel Xeon(R) E5-2690 CPU and 128GB memory. We kept a snapshot of the model over five different number of epochs: 10, 20, 30, 40, and 50. It took about 30 minutes to train an epoch and 25 hours for the entire training period. For new program generation, as described in *Design*, we used different sampling methods and various generation strategies to generate new C programs. The newly-generated programs are still based on the original training data; in another word, we used the original C programs as the seeds from which we randomly picked prefix sequences. By inserting new lines or replacing lines with new lines into a seed, we can get new programs. Because the newly-generated part will introduce new identifiers, new branches, new functions, etc., it will make the control-flow of the newly generated program more complicated and thus enhance the testing efficacy.

In our study, we use three metrics to measure the effectiveness of DEEPFUZZ:

- **Pass rate** is the metric to measure the ratio of syntax valid program among all of the newly generated C programs. The *Sequence-to-Sequence* model will presumably encode language patterns of C into the neural network. Therefore, the pass rate will be a good indicator of how well this network is trained over the input sequences. We use the command line of `gcc` to parse a newly generated program and if no error is reported, it indicates the syntactical correctness of this program.
- **Coverage** is a specific measurement for testing. Intuitively, the more code is covered by the tests, the more certainty we assure the completeness of testing. There

[2]We use lines of code instead of C syntactic objects such as statements since we treat C programs purely as sequences of characters.

[3]`https://github.com/s3team/DeepFuzz`

are three kinds of coverage information we collect during our analysis: line coverage, function coverage, and branch coverage. We use `gcov`, a command line tool supported by `gcc` to collect the coverage information.

- **Bug** detection is the goal of testing. For compiler testing, by feeding more programs to the compilers in different optimization levels, it is expected to trigger bugs like crashes or other code errors. As a self-protection mechanism, compilers like GCC and Clang/LLVM have defined a special kind of error called "internal compiler error". This error indicates the problem of the compiler itself during the compilation process and the error message will help us to find bugs in the compilers.

Pass rate

Pass rate is the ratio of generated syntax valid programs over the complete set of newly generated programs. It is an indicator of how well the C language patterns are encoded in the proposed *Sequence-to-Sequence* model. In our evaluation, specifically, we will analyze how the pass rate varies with the number of epochs of training, different sampling methods, and different generation strategies.

Epoch. An epoch is defined as an iteration of the learning algorithm to go over the complete set of the training data. We trained the model for a total of 50 epochs and we took a snapshot of the model at different epochs: 10, 20, 30, 40, 50 and applied the models for new C program generation. We tried the process for all the three sampling methods under the generation strategy *G1*.

Result: Figure 2 shows the result.

- The pass rate increases with more training from 10 to 30 epochs. The drop of pass rate after 30 epochs may be a result of overfitting.
- The best pass rate for all sampling methods is achieved at 30 epochs training. The highest pass rate is 82.63%.

Sampling. We have adopted different sampling methods after training the model. As we proposed, a sampling method decides how a new character is chosen based on the predicted distribution and it can affect the pass rate. Therefore, we recorded the pass rate of the newly generated 10,000 programs based on the seed programs under different sampling methods: *NoSample*, *Sample* and *SampleSpace*.

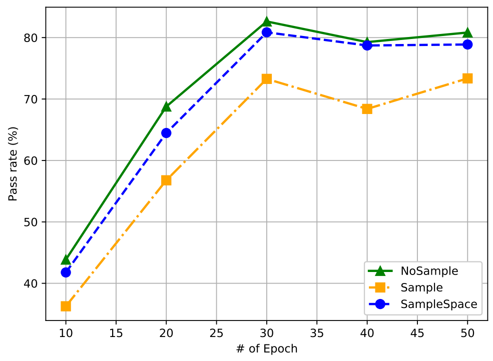


Figure 2: Pass rate for different sampling methods

Result: Figure 2 shows the result. Note, this experiment is conducted under the generation strategy *G1*.

- For all the sampling methods, the pass rate increases within 30 epochs of training and after that, there is a small drop.
- Comparing the pass rate for all the three sampling methods, *NoSample* achieves a better pass rate for every snapshot model than the other two methods *Sample* and *SampleSpace*. The highest pass rate is 82.63%.

Generation Strategy. To generate new programs, we have introduced three generation strategies: *G1)* insert two new lines at one location, *G2)* insert two new lines at different locations, and *G3)* replace two new lines. The newly generated lines are based on the prefix sequences selected in the seed programs. To analyze how the pass rate changes with different generation strategies, we recorded the result of performing program generation using a trained model after 30 epochs. In addition, we used *NoSample* in this experiment.

	Generation Strategy	**Pass rate (%)**
NoSample	G1	82.63
	G2	79.86
	G3	73.23

Table 1: Pass rate of 10,000 generated programs

Result: Table 1 shows the result.

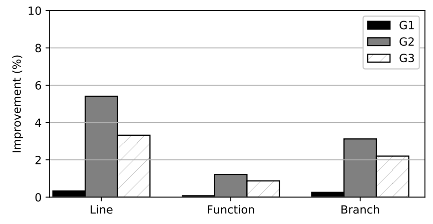
- The pass rate for the three generation strategies are 82.63%, 79.86%, and 73.23%, respectively. Comparing pass rate under these three different generation strategies, we conclude that *G1* performs the best in terms of the pass rate under *NoSample*.
- The result for *G1* and *G2* are similar in term of the pass rate which is higher than the pass rate for *G3*. The reason is probably that chopping out lines will introduce unbalanced statements, such as unclosed parenthesis, brackets, or curly brackets.

Coverage

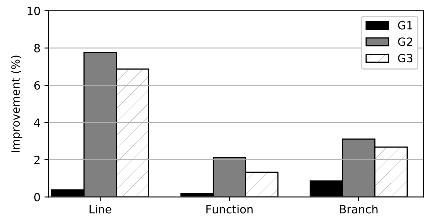
In addition to the pass rate, as described at the beginning of this section, because we are conducting testing, coverage information is another important metric. In this part, we analyzed how coverage improvements (line, function, branch) are achieved with different sampling methods and generation strategies.

Sampling. To compare the coverage improvements, we recorded the coverage information, including how many lines, functions, and branches are covered with the original seed test suite (10,000) plus the newly generated test suite (10,000) for both GCC-5 and Clang-3. In addition, to analyze how sampling methods can influence the coverage improvements, we recorded the coverage improvement percentages under different sampling methods.

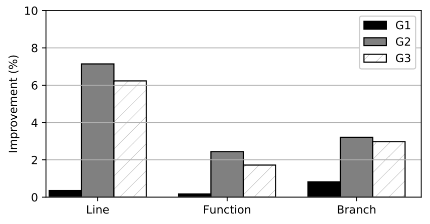
Result: The coverage improvement information is shown in Table 2 with the augmented test suite of 10,000 newly generated C programs from DeepFuzz on GCC-5 and to compare the metrics, we also present it in Figure 3.



(a) NoSample



(b) Sample



(c) SampleSpace

Figure 3: Coverage improvements for different sampling methods

		Line Coverage	**Function Coverage**	**Branch Coverage**
G1	NoSample	0.33%	0.08%	0.26%
	Sample	0.38%	0.19%	0.86%
	SampleSpace	0.36%	0.17%	0.82%
G2	NoSample	5.41%	1.22%	3.12%
	Sample	7.76%	2.13%	3.11%
	SampleSpace	7.14%	2.44%	3.21%
G3	NoSample	3.32%	0.87%	2.20%
	Sample	6.87%	1.33%	2.68%
	SampleSpace	6.23%	1.72%	2.97%

Table 2: Coverage improvements with 10,000 generated programs

- Among the three different sampling methods, *Sample* achieves the best performance in terms of line, function and branch coverage improvements. For example, under the generation strategy *G2*, the line coverage improvement for *NoSample*, *Sample* and *SampleSpace* is 5.41%, 7.76% and 7.14%, respectively.
- The coverage improvement patterns for different generation strategies are similar across different sampling methods. *G2* is always the best and *G1* is always the worst among the three. In another word, the performance of sampling methods is slightly correlated with generation strategies.

Generation Strategy. In addition to the sampling methods, we are also interested in how these three different coverages are improved under different generation strategies.

Result: Figure 3 shows how coverage improves using *G1*, *G2*, and *G3*.

- Comparing the coverage improvements under the three different generation strategies, *G2*, which is to insert two new lines at different locations, in most cases, achieves the best performance in terms of the line, function and branch coverage improvements.
- Comparing with sampling methods, the adoption of generation strategies is a more influential factor for coverage improvement. For instance, under *SampleSpace*, the function coverage improvement percentages for the three generation strategies are 0.17%, 2.44% and 1.72%. The coverage improvement increases 42 times after changing from *G1* to *G2*.
- *G2* and *G3* perform similarly in term of coverage improvement which is much higher than *G1*.

Overall. To demonstrate how our tool performs on compiler fuzzing, we compared DEEPFUZZ with a well-designed practical tool for compiler testing. Csmith (Yang et al. 2011) is a tool that can generate random C programs. To make a fair comparison, we recorded the coverage improvements of Csmith and DEEPFUZZ by both augmenting the GCC and LLVM test suites with 10,000 generated programs in Table 3.

Note that we use *Sample* as our sampling method and *G2* as our generation strategy when conducting this analysis. We also documented coverage improvements during the process of program generation in Figure 4. It demonstrates how the line, function, and branch coverages are improved with the increasing number of new tests.

		Line Coverage	**Function Coverage**	**Branch Coverage**
GCC	original	75.13%	82.23%	46.26%
	Csmith	75.58%	82.41%	47.11%
	% change	+0.45%	+0.18%	+0.85%
	DEEPFUZZ	82.27%	84.76%	49.47%
	% change	+7.14%	+2.44%	+3.21%
	absolute change	+23,514	+619	+16,884
Clang	original	74.54%	72.90%	59.22%
	Csmith	74.69%	72.95%	59.48%
	% change	+0.15%	+0.05%	+0.24%
	DEEPFUZZ	79.89%	74.56%	66.79%
	% change	+5.35%	+1.66%	+7.57%
	absolute change	+23,661	+2,456	+26,960

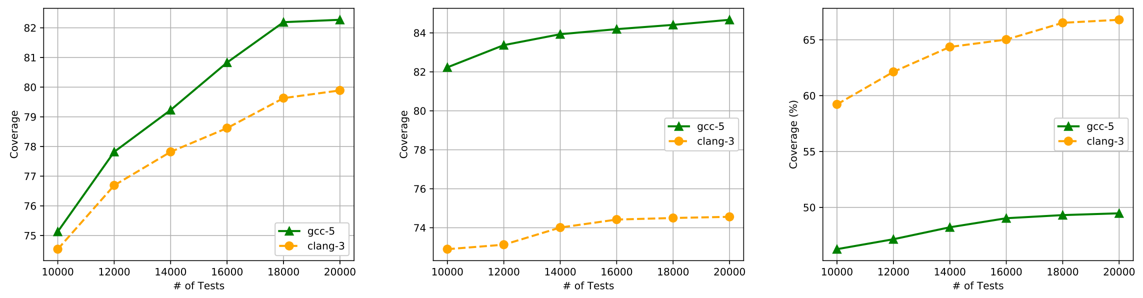
Table 3: Augmenting the GCC and LLVM test suites with 10,000 generated programs

Result:

- Csmith improved the coverage less than 1% for all the cases while DEEPFUZZ improves the coverage of line, function, and branch by 7.14%, 2.44%, and 3.21%, respectively. DEEPFUZZ achieves better coverage improvement than Csmith.
- The performance of the coverage improvement pattern for DEEPFUZZ is similar over GCC-5 and Clang-3.

New bugs

Using different generation strategies and sampling methods, based on the seed programs from the GCC test suite, we can generate new programs. Because we aim at compiler fuzzing, the number of bugs detected is an important indicator of the efficacy of DEEPFUZZ. During our preliminary



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

Figure 4: Coverage improvement with the new tests generated

study, we caught *8* newly confirmed GCC bugs and we will elaborate on two bugs that we detect with more details.

GCC Bug 84290: This is a bug we reported. DEEPFUZZ generate the two new lines (line 5 and line 6), which triggered an internal compiler error of the built-in function __atomic_load_n. The error is triggered because that the first argument of this function should be a pointer, but it points to an incomplete type. This error is fixed and a new test (atomic-pr81231.c) is added to the latest test suite in GCC. This detected bug shows the importance of using the syntactically well-formed but semantically nonsense tests for compiler testing.

```
1  double f () {
2    double r;
3    asm ("mov␣%S1,%S0;␣mov␣%R1,%R0" : "=r" (r) : "i" (20));
4    asm ("mov␣%S1,%S0;␣mov␣%R1,%R0" : "+r" (r) : "i" (20.));
5    __atomic_load_n ((enum E *) 0, 0);
6    ;
7    return r;
8  }
```

GCC Bug 85443: This is a bug we reported. DEEPFUZZ generates the two new lines (line 5 and line 6), which introduced a new crash. The generated _Atomic is a keyword for defining atomic types and the assignment on line 6 triggers the segmentation fault. This is a newly confirmed bug on GCC-5 and has been fixed in the latest version. This detected bug by DEEPFUZZ again shows the importance of using the syntactically well-formed but semantically nonsense tests for compiler testing.

```
1  char acDummy[0xf0] __attribute__ ((__BELOW100__));
2  unsigned short B100 __attribute__ ((__BELOW100__));
3  unsigned short *p = &B100;
4  unsigned short wData = 0x1234;
5  _Atomic int i = 3;
6  int a1 = sizeof (i + 1);
7  void Do (void) {
8    B100 = wData;
9  }
10 int main (void) {
11   *p = 0x9876;
12   Do ();
13   return (*p == 0x1234) ? 0 : 1;
14 }
```

Limitations

Observing the generated programs, we noticed that many ill-formed generations are caused by expected expressions. To be more specific, this error message denotes the errors like unbalanced parenthesis, brackets, or curly brackets. We conclude two main reasons that account for this problem: lack of training and loss of global information.

For the first reason, the training data is abundant but it still lacks enough repeated patterns in the current training dataset for training a good generative model. In our future work, we can create a larger training dataset by enumerating all the structures in the original test suites with new variable or function names. On the other hand, because the generation is based on the prefix sequences, it will lose some global information which is out of the scope of the prefix sequence. To handle this problem, we either increase the length of the training sequence to ensure that enough information is captured, or we can use some heuristics to help with model training. The former method may cause less diversity in the generated program and the latter one requires the assistance of static program analysis.

Additionally, our proposed method is based on a character-level *Sequence-to-Sequence* model. We provide a sequence of characters for the current model which requires a lot of effort in dealing with the token-level syntax. It hurts the training scalability and pass rate as well. In C, there are less than 32 keywords and over 100 build-in functions. Both the pass rate and scalability will be increased if we perform token-level sequence prediction over a *Sequence-to-Sequence* model.

Related Work

AI-based applications for software security and software analysis are widely discussed over the years (Zamir, Stern, and Kalech 2014; Elmishali, Stern, and Kalech 2016; Nath and Domingos 2016). Neural network based models dominant a variety of applications and there has been a tremendous growth in interest in using them for program analysis (Allamanis and Sutton 2013; Nguyen et al. 2013) and synthesis (Lin et al. 2017; Devlin et al. 2017). Recurrent neural networks especially *Sequence-to-Sequence*-based models have been developed for learning language models of source code from a large code corpus and then using these models for several applications, such as learning natural coding conventions, code suggestions, and auto-completion and repairing syntax errors (Bhatia and Singh 2016; Hindle et al. 2012). It has been proven efficient, especially when a large amount of data is provided, in improving the system efficacy as well as saving human labor. Additionally, RNN-based models are applied for grammar-based fuzzing (Godefroid, Peleg, and Singh 2017; Cummins et al. 2018) which learns a generative model to produce PDF files to fuzz the PDF parser.

Conclusion and Future Work

Compiler testing is critical for assuring the correctness of computing systems. In this paper, we proposed an automatic grammar-based fuzzing tool, called DEEPFUZZ, which learns a generative recurrent neural network that continuously generates syntactically correct C programs to fuzz the off-the-shelf production compilers. DEEPFUZZ generated 82.63% syntax valid C programs and improved the testing efficacy in regards to the line, function and branch coverage. We also found new bugs which are actively being addressed by developers.

Acknowledgement

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research. This research was supported in part by the National Science Foundation (NSF) grants CNS-1652790 and the Office of Naval Research (ONR) grants N00014-13-1-0175, N0001416-1-2265, N00014-16-1-2912, and N00014-17-1-2894.

References

Allamanis, M., and Sutton, C. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proc. of the 10th Working Conference on Mining Software Repositories*.

Bhatia, S., and Singh, R. 2016. Automated Correction for Syntax Errors in Programming Assignments Using Recurrent Neural Networks. *arXiv preprint arXiv:1603.06129*.

Bird, D. L., and Munoz, C. U. 1983. Automatic generation of random self-checking test cases. *IBM Systems Journal* 22(3).

Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proc. Empirical Methods in Nat. Lang. Proc.*

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Clang: a C language family frontend for LLVM. 2018. clang.llvm.org.

Cleve, H., and Zeller, A. 2005. Locating causes of program failures. In *International Conference on Software Engineering (ICSE)*.

Cummins, C.; Petoumenos, P.; Murray, A.; and Leather, H. 2018. Compiler fuzzing through deep learning. In *ISSTA'18*.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; rahman Mohamed, A.; and Kohli, P. 2017. RobustFill: Neural program learning under noisy I/O. In *Int'l Conf. on Machine Learning*.

Dewey, K.; Roesch, J.; and Hardekopf, B. 2014. Language fuzzing using constraint logic programming. In *ASE'14*.

Elmishali, A.; Stern, R.; and Kalech, M. 2016. Data-augmented software diagnosis. In *AAAI-16*.

GCC, the GNU Compiler Collection. 2018. gcc.gnu.org.

Godefroid, P.; Kiezun, A.; and Levin, M. Y. 2008. Grammar-based whitebox fuzzing. In *PLDI'08*.

Godefroid, P.; Peleg, H.; and Singh, R. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *ASE'17*.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the Naturalness of Software. In *ICSE'12*.

Hoare, T. 2003. The verifying compiler: A grand challenge for computing research. In *International Conference on Compiler Construction*.

Klein, G.; Kim, Y.; Deng, Y.; Senellart, J.; and Rush, A. M. 2017. OpenNMT: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*.

Le, V.; Afshari, M.; and Su, Z. 2014. Compiler validation via equivalence modulo inputs. In *PLDI'14*.

Le, V.; Sun, C.; and Su, Z. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA'15*.

Leroy, X., and Grall, H. 2009. Coinductive big-step operational semantics. *Information and Computation* 207(2).

Leroy, X.; Blazy, S.; Kästner, D.; Schommer, B.; Pister, M.; and Ferdinand, C. 2016. CompCert—a formally verified optimizing compiler. In *ERTS 2016: The 8th European Congress on Embedded Real Time Software and Systems*.

Lidbury, C.; Lascu, A.; Chong, N.; and Donaldson, A. F. 2015. Many-core compiler fuzzing. In *PLDI'15*.

Lin, X. V.; Wang, C.; Pang, D.; Vu, K.; and Ernst, M. D. 2017. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA.

Nath, A., and Domingos, P. M. 2016. Learning tractable probabilistic models for fault localization. In *AAAI-16*.

Necula, G. C. 2000. Translation validation for an optimizing compiler. In *Proc. PLDI'00*.

Nguyen, T. T.; Nguyen, A. T.; Nguyen, H. A.; and Nguyen, T. N. 2013. A statistical semantic language model for source code. In *Proc. FSE'13*.

of the International Organization for Standardization (ISO), J. T. C. 2011. Sc22/wg14. iso/iec 9899: 2011. *Information technology, Programming languages, C*.

Shi, X.; Padhi, I.; and Knight, K. 2016. Does string-based neural MT learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*.

Sun, C.; Le, V.; Zhang, Q.; and Su, Z. 2016. Toward understanding compiler bugs in GCC and LLVM. In *ISSTA'16*.

Sutskever, I.; Martens, J.; and Hinton, G. E. 2011. Generating text with recurrent neural networks. In *Proc. ICML'11*.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*.

Yang, X.; Chen, Y.; Eide, E.; and Regehr, J. 2011. Finding and understanding bugs in C compilers. In *PLDI'11*.

Zalewski, M. 2015. American fuzzy lop (AFL) fuzzer (2015).

Zamir, T.; Stern, R. T.; and Kalech, M. 2014. Using model-based diagnosis to improve software testing. In *AAAI-14*.