DSmith: Compiler Fuzzing through Generative Deep Learning Model with Attention

Haoran Xu, Yongjun Wang*, Shuhui Fan, Peidai Xie, Aizhi Liu†
College of Computer, National University of Defense Technology, China
†*Institute of War, Academy of Military Sciences, China*
{xuhaoran12, wangyongjun, fanshuhui18}@nudt.edu.cn, xpd2002@126.com, ams_newage@163.com

Abstract*—Compiler fuzzing is a technique to test the functionalities of compiler. It requires well-formed test cases (i.e., programs) that have correct lexicons and syntax to pass the parsing stage of a compiler. Recently, advanced compiler fuzzing methods generate effective test cases by deep neural networks, which learn the language model of regular programs to guarantee test case quality. However, most of these methods fail to capture long-distance dependencies of syntax (e.g., paired curly braces) in a program. As a result, they may generate test cases with syntax errors, which cannot pass the parsing stage to test the compiler functionality. In this paper, we propose a framework, namely DSmith, to capture long-distance dependencies of syntax for a robust test case generation. Specifically, DSmith memorizes the hidden state of each token in a program and leverages the interactions of these hidden states to embed the long-distance dependencies between tokens. It then adopts an encoder-decoder architecture with the embedding of these long-distance dependencies to build a language model of regular programs. Finally, DSmith uses the built language model to generate test cases according to four novel generation strategies, which significantly increase the diversity of test cases. Extensive experiments show that DSmith increases the parsing pass rate of the generated programs by an average of 19% and significantly improves the code coverage of the compiler, compared with state-of-the-art methods. Benefiting from the high pass rate and broad code coverage, DSmith has found eleven brand new bugs in currently supported GCC compiler versions.

***Index Terms*—fuzzing, compiler, neural network, attention, syntax**

I. Introduction

Compiler is one of the most central components in a software development tool chain. Compilers like GCC and clang/llvm are not only typical representatives of open source software but also important basic software of the open source community. Various efforts have been invested to improve the correctness and reliability of compilers. Fuzzing is an effective technique for finding bugs and security vulnerabilities in compilers. It is an automated testing technique that aims to trigger unintended compiler behaviors by feeding a large number of test programs to the target compiler. The quality of the input programs significantly influence the effectiveness of the fuzzers.

Effective compiler fuzzing requires well-formed test programs with correct lexicons and syntax to pass the parsing stage of a compiler, after which the complex functionalities can be tested. To comprehensively discover bugs, fuzzing methods need to generate test cases with a high parsing pass rate.

Recently, lots of methods adopt deep neural networks to learn correct syntax from regular programs. For example, Liu et al. [1] used a sequence-to-sequence (seq2seq) architecture based on recurrent neural networks (RNN) to build a generative model for fuzzing C program generation. These deep neural network-based fuzzing methods can automatically learn syntax from a large number of regular programs. Accordingly, these methods do not rely on complex syntax rules provided by human, which is more time-saving and labor-saving compared with the traditional rule-based fuzzing methods, such as CSmith [2].

Although the existing deep neural network-based methods learn the syntax automatically, most of them may fail to generate programs with correct syntax when facing the long-distance dependencies problems that commonly exist in program syntax. During the generation process, these methods receive a code sequence as the conditional input and predict subsequent codes. They then append the generated code to the end of the input to form a new code sequence, which will be used as the conditional input to generate the following code. These methods repeat this generation and appending process until generating a complete program. However, when the length of the program is long (e.g., over 100 tokens), the existing methods are hard to generate the correct syntax, such as curly braces, at the end of the generated program because they overlook long-distance dependencies. For example, a side of a brace at the end of a program may depend on another side of the brace at the beginning of the program. When existing several opening braces in a long conditional input, the existing methods may not generate the correct number of closing braces for matching.

In this paper, we propose a novel fuzzing framework, namely DSmith, to tackle the long-distance dependencies problem in the existing compiler fuzzing methods. The DSmith framework memorizes the hidden state of each token (e.g., keyword, identifier, variable, and constant) in a program and leverages the interactions of these hidden states to embed the long-distance dependencies. Accordingly, it can generate an appropriate token that depends on other tokens even their locations have a long distance. To achieve this goal, DSmith introduces a long short-term memory (LSTM) unit [3] and an attention mechanism to memorize hidden state and capture

*Corresponding author.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

```c
int sub_0 ( void ) {
  int var_0 [ 4 ] = { 30 , 2 , 10 , 5 } ;
  int var_1 [ 2 ] ;
  int var_2 , var_3 , var_4 ;

  for ( var_2 = 2 ; var_2 < 4 ; var_2 ++ ){
    if ( var_0 [ var_2 ] < var_5 ){
      var_1 [ var_4 ] = var_0 [ var_2 ] ;
      var_4 = 0 ;
      var_5 = var_1 [ 0 ] ;
      for ( var_3 = 1 ; var_3 < 2 ; var_3 ++ )
      if ( var_1 [ var_3 ] > var_5 ){
        var_4 = 0 ;
      }
// The above code is used as conditional input
// Following code is produced by model
    }
  }
}
}
}
}
```

Listing 1. Program produced by seq2seq model

```c
int sub_0 ( void ) {
  int var_0 [ 4 ] = { 30 , 2 , 10 , 5 } ;
  int var_1 [ 2 ] ;
  int var_2 , var_3 , var_4 ;

  for ( var_2 = 2 ; var_2 < 4 ; var_2 ++ ){
    if ( var_0 [ var_2 ] < var_5 ){
      var_1 [ var_4 ] = var_0 [ var_2 ] ;
      var_4 = 0 ;
      var_5 = var_1 [ 0 ] ;
      for ( var_3 = 1 ; var_3 < 2 ; var_3 ++ )
      if ( var_1 [ var_3 ] > var_5 ){
        var_4 = 0 ;
      }
// The above code is used as conditional input
// Following code is produced by model
    }
  }
  return 0;
}
```

Listing 2. Program produced by DSmith

Fig. 1. The two programs shown in this figure are the test programs generated by the seq2seq model and DSmith respectively. The seed program comes from the GCC test suite and has been pre-processed. The details of the preprocessing are described in the third part of the paper. Taking the code from lines 1 to 17 as conditional inputs for the generative model, we use the seq2seq model and DSmith to predict the subsequent code. As shown in Listing 1, the code sequence predicted by the seq2seq model contains too many closing brackets, which results in a syntax error and the program cannot pass through the parsing of compilers. The code sequence predicted by DSmith matches the conditional input well, guaranteeing the generated program can pass the parsing stage successfully.

their interactions, respectively. It then adopts an encoder-decoder architecture [4] with the embedding of these long-distance dependencies to build a language model of regular programs. Based on the built language model, DSmith can generate programs given a conditional input (e.g., a piece of a program). To increase the fuzzing program diversity for broader coverage, we further propose four different program generation strategies. According to these strategies, DSmith can generate various effective test programs by its built language model.

To demonstrate the effectiveness of DSmith, we adopt DSmith for C compiler fuzzing. We illustrate this effectiveness on a toy example compared with a seq2seq-based fuzzing method [1] in Fig. 1. As shown in Fig. 1, the code sequence generated by the seq2seq-based method contains too many closing brackets, which results in a syntax error, and thus, this code cannot pass the parsing stage. In contrast, the code sequence generated by DSmith matches the conditional input well, guaranteeing the test program can pass the parsing stage successfully.

We summarize the contributions of this paper as follows:

- We capture long-distance syntax dependencies in programming language to further generate effective compiler fuzzing inputs. We use the attention mechanism to capture the long-distance syntax dependencies in programs. Experiments show that programs generated by DSmith are more syntactically correct.
- We present DSmith, a framework consists of attentive learning a language model of C programs, automatic generation of new test cases, and performing fuzz testing. According to the four novel generation strategies, DSmith can generate various effective test programs by its built language model. It has been applied to the fuzzing of GCC on all currently supported versions. We have found and reported eleven bugs.

Extensive experiments show that: (1) the parsing pass rate of the test cases generated by DSmith can reach a maximum of 78%, which improves by an average of 19% compared with the state-of-the-art methods; (2) the test cases generated by DSmith cover more compiler code based on the same seed programs as other competitors; and (3) DSmith has found eleven brand new bugs in four GCC compiler versions.

II. Background

A. Recurrent Neural Network and Sequence-to-Sequence Models

In recent years, deep learning has achieved great success in generation tasks such as machine translation [4] and language modeling [5]. Compared with traditional rule-based and probabilistic model-based methods, deep neural networks-based methods do not rely on expert engineering and have better scalability.

Many generation tasks involve sequential data. Recurrent neural networks (RNNs) are designed to model sequential data [6]. When operating on a variable-length input sequence $X = (x_1, \ldots, x_T)$, RNNs use a hidden state to carry information to later time steps. At each time step, the RNNs receive

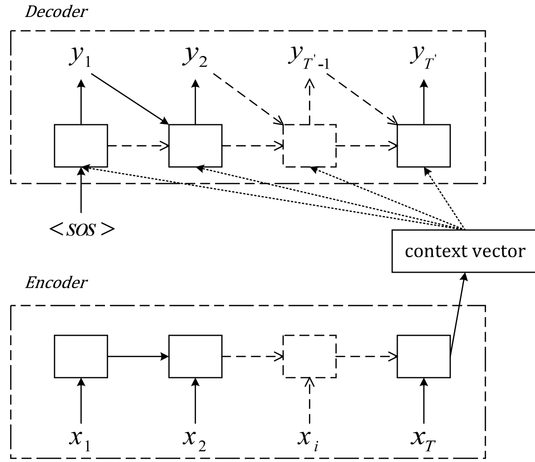


Fig. 2. encoder-decoder architecture

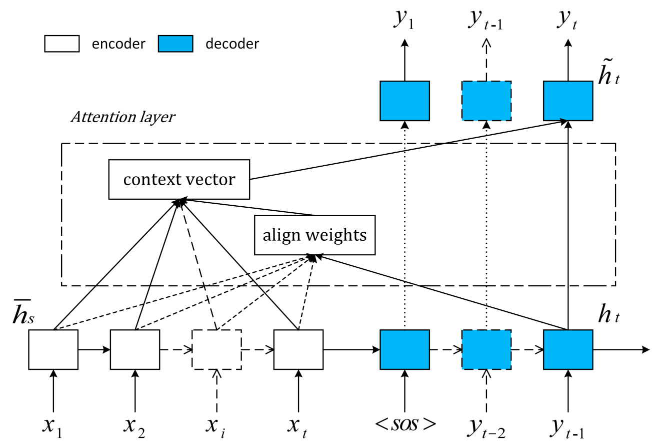


Fig. 3. seq2seq model with attention mechanism

an input and update their hidden state. The hidden state h_t and optional output are updated by

$$h_t = f_h(x_t, h_{t-1}) \tag{1}$$

$$y_t = f_o(h_t) \tag{2}$$

where f_h is a non-linear activation function, and f_o is the output function. Different gating mechanisms have been proposed to improve the performance. LSTM was introduced by Hochreiter et al. [3] and was better at learning long-term dependencies. Cho et al. [7] proposed a gated recurrent unit to adaptively capture dependencies of different time scales.

Encoder-decoder architecture [4] with RNNs was introduced for machine translation. The architecture has become an effective model for seq2seq prediction, since it has the ability to handle variable-length input and output sequences. The schematic view of the encoder-decoder architecture is shown in Fig. 2.

There are two RNNs in encoder-decoder architecture. The first one is the encoder, which learns to encode a variable-length inputs into a context vector representation. The decoder decodes the vector back into a variable-length output.

The encoder reads each symbol of the inputs x_t and updates the hidden state h_t sequentially. By the time the last symbol is processed, the final hidden state contains all the information of the input sequence. The decoder is trained to predict the next symbol y_t given the hidden state h_t. Both y_t and h_t are conditioned on y_{t-1} and the context vector c

$$h_t = f(h_{t-1}, y_{t-1}, c) \tag{3}$$

The encoder-decoder model learns a probability distribution over a sequence of symbols to predict the next symbol. The conditional distribution of the next symbol is

$$P(y_t|y_{t-1}, y_{t-2}, \ldots, y_1, c) = g(h_t, y_{t-1}, c) \tag{4}$$

where g is usually a softmax function to produce valid probabilities.

B. Attention mechanism

A potential issue with the encoder-decoder architecture is that the encoder compresses all necessary information of the input sequence into a fixed-length vector representation, called context vector c. When decoding, the decoder always uses the same context vector for each output step. The fixed-length vector becomes a bottleneck in this encoder-decoder architecture. Cho et al. [7] showed that the performance of the basic encoder-decoder model decreases with the increase of sequence length.

Attention mechanism was first introduced in natural language processing for machine translation task by Bahdanau et al. [8]. Many recent works showed that attention could be successfully used in various tasks. These include, but not limited to, image caption [9], summarization [10], and document classification [11].

As a component of the network architecture, attention mechanism is based on the concept of assigning higher weights to important and relevant elements in the input sequence to enhance the accuracy of output prediction. Bahdanau et al. proposed a model which stored all the encoder RNN's outputs and used them together with the decoder RNN's state h_{t-1} to compute the context vector. The context vector was then used to compute the state h_t. Luong et al. [12] presented a generalization of the attention mechanism. In this instance h_t rather than h_{t-1}, along with the outputs of the encoder was used to compute a context vector. This vector was concatenated with h_t to make the next prediction. Recently, Yu et al. [13] showed the ability of a seq2seq model with attention mechanism to learn context-free language grammars.

An overview of the seq2seq model with attention mechanism is provided in Fig. 3, where $\overline{h}_s$ denotes all encoder hidden states, h_t denotes the decoder state at time step t. Unlike using a fixed context vector c in traditional encoder-decoder approach, this attentional model infers a alignment weight based on h_t and $\overline{h}_s$. Source-side context vector is then computed as a weighted average according to the alignment weight over all the source hidden states. Lastly, we get a new

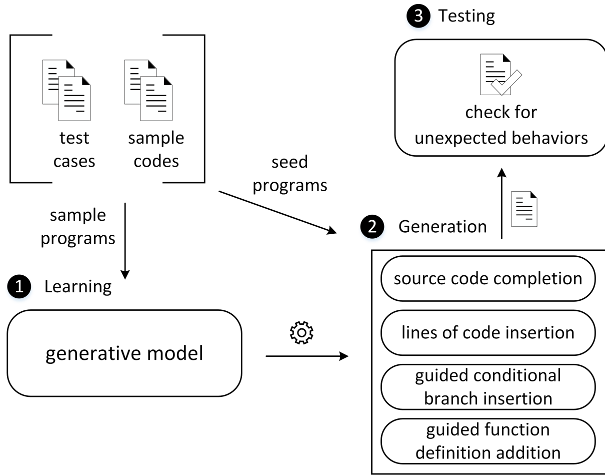


Fig. 4. An overview of DSmith

form of y_t:

$$y_t = f_{decoder}(c_t, h_t) \tag{5}$$

III. DSmith

In this section, we provide an overview on the overall design and implementation details of DSmith.

A. Design Overview

The architecture overview of DSmith is depicted in Fig. 4. There are three main components in DSmith. The learning component learns the language model from many regular programs that are syntactically correct. The generation component generates as many valid programs as possible according to the four strategies. The generation component then sends the generated programs to the testing component. The testing component checks whether these new test cases trigger bugs in target compiler.

B. Attentive Neural Network Model

a) Dataset: We use the test suite in the compiler source releases as the original dataset. We collect grammatically correct programs from these test suites.

b) Preprocessing: In order to remove the influence of noise factors, we preprocess these test cases. We remove all the comments, and expand all macros. Considering that the inclusion of header files will introduce a large number of duplicate standard library headers, we remove the source files containing `include` preprocessing directive from the dataset. The processed dataset has about 14000 test programs.

c) Tokenization and simplification: The smallest individual grammatical unit in C programs is token. As we aim to build the language model at token level, there are some challenges to overcome. Due to the diversity of identifier naming in programming languages, the vocabulary size would be very large if we use the token sequence in the dataset directly as input.

To facilitate the model to learn the syntax of C programming language, we further process the source files in the dataset, trying to reduce the number of tokens.We simplify the number of tokens in two ways. First, we rename identifiers in the programs. Renamed identifiers include function name, variable name, parameter name, structure name, enumeration name, union name, type definition name, and field name inside a composite data type. For example, for all function names in a source file, we name them from `sub_0`, and rename them one by one according to the pattern of `sub_n`. For all the variable names in a source file, we name them from `var_0`.Second, we rename the literals in the programs. Renamed literals include character literals, string literals, and floating point literals. We rename character literals to a same value of `"C"`, string literals to a same value of `"STRING"`, and floating point literals to a fixed floating point number like `1.1`. Changing the value of literals might change the execution flow of the original program, we think it is a tradeoff to reduce the vocabulary. Finally, we unify the use of blank characters. Only a single space is used as blank character. After this, we process each source file into 1 line of code.

d) Attention mechanism: We extend the encoder-decoder architecture with attention mechanism in DSmith. The neural network we employ in the encoder and decoder model is LSTM.

After processing all the input data, the encoder generates hidden state for each element in the input sequence. Unlike traditional seq2seq model which only use the hidden state at the final time step, we collect all the hidden states. In the decoder part, the previous decoder output and hidden state is passed through the decoder LSTM, and a new hidden state h_t for that time step is produced after that. We then calculate alignment scores using the new decoder hidden state and all encoder hidden states:

$$score(h_t, \overline{h}_s) = {h_t}^\top W_a \overline{h}_s \tag{6}$$

where $\overline{h}_s$ denotes each encoder hidden state.

The alignment scores for every encoder hidden state are combined and represented to a single vector. We use the softmax function to normalize it into a probability distribution.

$$a_t(s) = align(h_t, \overline{h}_s) = \frac{exp(score(h_t, \overline{h}_s))}{\sum_{s'} exp(score(h_t, \overline{h}_{s'}))} \tag{7}$$

where $a_t(s)$ is the alignment weight of sth location at step t in the input sequence.

The alignment weights and their respective encoder hidden states are multiplied to form the context vector. We then use the context vector concatenated with decoder hidden state to produce an attentional hidden state passed through a fully connected layer.

$$\widetilde{h}_t = tanh(W_c[c_t; h_t]) \tag{8}$$

where c_t denotes the source-side context vector, $\widetilde{h}_t$ denotes the attentional hidden state.

Lastly, we use the attentional hidden state to produce the target output:

$$p(y_t|y_{<t}, x) = softmax(W_s\widetilde{h}_t) \tag{9}$$

Considering the complexity and efficiency of the generative model, DSmith has two layers of LSTM networks at the encoder and decoder side, and the number of hidden units in each layer is set to 512. Embedding size is set to 128. We set the dropout rate of 0.2.

C. Generating new test programs

Once the model training is finished, we send token sequences to the model and ask the model to predict subsequent tokens of the program. The generated outputs are then added to the code prefix and this process is repeated several times. This loop can generate a code sequence of any length.

During this process, how to select the conditional token sequence and sample the next token is crucial. We randomly select program fragments from a seed program as input to the model instead of using a fixed starting prefix. At this stage, we still use the test cases we used when training the model as seed programs. Our goal is to increase the diversity of generated programs.

1) Generation strategy: To make full use of generative models to produce diverse test programs, we propose four generation strategies: source code completion, lines of code insertion, guided conditional branch insertion, and guided function definition addition. Generation strategies are flexible, we focus on applying and testing these four strategies.

Source code completion. This strategy refers to selecting a starting point of code completion from the source file, discarding the code after the position, and using the code sequence before that position as the conditional input to the model for new code generation. In this way, the proportion of newly generated code in the test program may be high, thus giving the model a large room for creativity.

Lines of code insertion. This strategy refers to selecting a code insertion point from the source file, using the code sequence before that position as the model's input, and intercepting a certain number of lines of code from the model's output and inserting it back into that position. Because this strategy does not remove code from the source code, it is less disruptive to the seed program, making it easier to generate syntactically compliant test programs.

Guided conditional branch insertion. This strategy refers to selecting a code insertion point from the source file, adding the two tokens of `if` and the left parenthesis `(` to the code sequence after that position, and using them as input to the model. Finally, a certain number of lines of code are intercepted from the model's output and inserted back to that location. Since the two tokens have been added at the end of the prefix for guidance, the model will generate at least one conditional statement, thereby adding branches and new scopes to the seed program.

Guided function definition addition. This strategy refers to guiding the generative model to add function definitions to the seed program. In order to facilitate processing, we add a token sequence at the end of the source file to guide the generation of a new function, such as `void sub_x (` and `int sub_x (`. The value of `x` is determined by the number of functions defined in the seed program. We then pass the full code sequence to the model to generate subsequent code. We truncate a certain length from the model's output to construct new test programs. This strategy adds new function definitions and new scopes to seed programs.

For each generation strategy, we use the code sequence before the starting/insertion point as the starting prefix. The model generates new tokens until token `EOS` which marks the end of a source file. When the generation is completed, we select the truncation points from the output code sequence, and append/insert the part before the truncation point to the original insertion point. For the first three generation strategies, the starting/insertion points are not unique. We choose the position after the semicolon token as an optional insertion point. For the fourth generation strategy, we only add code sequence to the end of the seed program, so the insertion point is unique. For the first strategy, we append all the code before `EOS` token to the starting point, so the truncated position is unique. For the latter three strategies, we use a certain length from the generated sequence, and the truncated position is determined based on the tokens `;` or `}`.

2) Sampling method: We sample the next token from the probability distribution of the model's outputs. A simple method is greedy sampling, which always selects the next token with the greatest possibility. The token sequence produced by this approach is well-formed. However, the resulting program lacks variety. Also, the same conditional input produces deterministic output.

Another sampling method is stochastic sampling which randomly selects the next token based on the probability distribution. Rather than completely random sampling, a more reasonable way is to sample from a multinomial distribution. When sampling from a multinomial distribution, the output with the greatest probability can be selected with high probability. Moreover, there is a certain probability to explore other outputs. Temperature can be used to control the randomness of the stochastic sampling process. A lower temperature value means that the sampling will be more conservative and the generated program is more likely to be well-formed. A higher temperature will bring more variety, but will also bring some syntactically incorrect outputs. For fuzz testing, on the one hand, we need to produce well-formed input to pass the validity check of the parsing stage. On the other hand, we want to produce as diverse an output as possible to trigger unexpected behaviors. In DSmith, we use both of these two approaches.

IV. Experiments and Evaluation

In this section we assess the effectiveness of DSmith via an empirical analysis and demonstrate its usefulness in fuzzing real-world compilers.

Experiment Setup. All experiments were conducted on our high-performance workstation. The workstation has 2 Intel Xeon 4114 CPUs with 10 cores each, running a 64-bit Ubuntu 16.04 with kernel version 4.13.0-36. For the compilation we set a timeout of 60 seconds per source file.

Evaluation metric. We use three metrics to measure the effectiveness of DSmith: compilation pass rate of generated test programs, code coverage improvements of the target compiler, and unique bugs found in compilers.

The compilation pass rate of generated code is the proportion of syntax valid program among all generated test case. We use this metric to measure how well the model learns the syntax of the input data. When fuzzing compilers like GCC, the ability to generate syntax-compliant input files is critical. The test programs produced by the generative model conform to the syntax of the programming language, which means that the test case could pass the legitimacy check of the parsing stage, such as the lexical analysis and syntax analysis, so as to reach the more complex and vulnerable parts of the compiler.

For software testing, code coverage is a common evaluation indicator. It is a metric that describes the degree of which the source code of the program has been tested. Many popular fuzzers are guided by code coverage, like AFL [14], HonggFuzz [15], and libFuzzer [16]. Although our fuzzing approach is not coverage-guided, the coverage information can still help us evaluate and analyze our fuzzing process. We use gcov[1] and lcov[2] to gather the coverage information for analysis.

We apply DSmith on real-world compilers and use the number of found bugs as one of the criteria for the effectiveness of this tool. Compilers like GCC has defined a special kind of error called “internal compiler error”, commonly abbreviated as “ICE”. An internal compiler error is an error that occurs not due to erroneous source code, but rather due to a bug in the compiler itself. When the test input triggers a internal compiler error in GCC, GCC prints out the function stack to help the developer resolve the problem.

A. Compilation pass rate

The compilation pass rate indicates the effect of the generative model to learn patterns in C programming language. In this part, we will analyze how the attention mechanism, four generation strategies, and different sampling methods affect the compilation pass rate of the generated programs.

As long as no error occur during compilation, we assume that the test program has been compiled successfully. In our experiments of collecting pass rates, all the inputs that caused the compilation to fail are due to syntax errors in the generated test program, rather than triggering a compiler bug.

1) Comparison between different models and generation strategies: To compare the pass rates between different generation strategies, we use DSmith to generate 10000 test programs under different generation strategies, and compared the pass rates of these programs. To analyze the performance of DSmith, we use the seq2seq-based method [1], also token level, as the baseline. We use greedy sampling to perform this comparison.

The results are shown in the Fig. 5. In summary, the introduction of attention mechanism significantly increases the

[1]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[2]http://ltp.sourceforge.net/coverage/lcov.php

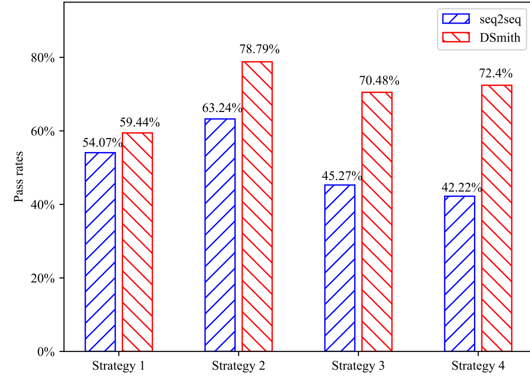


Fig. 5. Pass rate of different strategies

compilation pass rate of programs generated under each generation strategy. Especially for the fourth generation strategy, the pass rate of the program generated by DSmith is 30% higher than that of the ordinary seq2seq model.

For different generation strategies, strategy 2 has the highest compilation pass rate. This is because the way of inserting has less influence on the structure and logic of the original seed program, so it is easier to generate test programs that conform to the syntax. Besides, the insertion point of the original code sequence and the truncation point of the generated sequence are not unique. Therefore, the number of test programs that can be generated for each seed program is also greater. Among all strategies, the pass rate of the code generated by strategy 1 is the lowest. As we mentioned earlier, the generation method of source code completion provides a large creative space for the generative model. The generated test program may have a large proportion of new code. This makes the generated program prone to problems that do not meet the grammar specification.

2) Comparison between different sampling temperatures: Stochastic sampling can bring diverse output to generative models, which is important for fuzzing. To analyze the influence of the temperature value on the pass rate of the generated code during stochastic sampling, we select 5 different temperature values of 0.25, 0.5, 0.75, 1.0, and 1.25, to compare the pass rate of the program generated by DSmith with strategy 2.

The results are shown in the Fig. 6. As temperature values rise, the generative model becomes more aggressive. This may bring unexpected but useful test inputs to trigger compiler vulnerabilities. However it also makes it harder for programs generated by the DSmith to conform to the input syntax, thereby reducing the compilation pass rate of these test cases. When the temperature value is 0.5, the pass rate is reduced by about 9% compared to greedy sampling. When the temperature value increases to 1.25, the pass rate drops below 40%.

B. Code Coverage

In this part, we compare the effect of DSmith and seq2seq method on improving the code coverage of the compiler. And

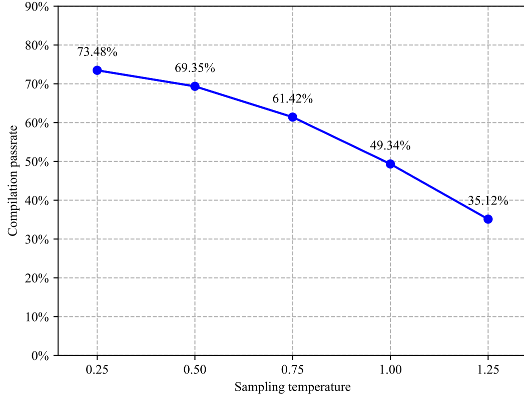


Fig. 6. Pass rate of different sampling temperatures

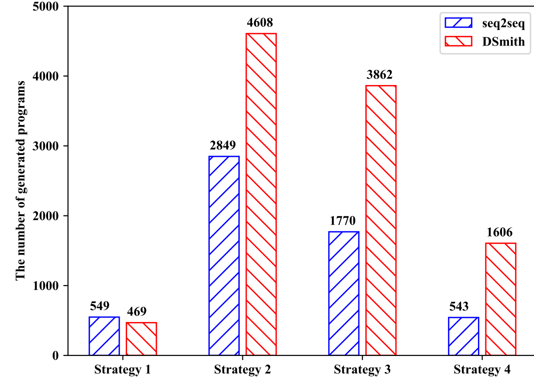


Fig. 7. Numbers of generated valid programs

we compare the effect of different generation strategies and different sampling methods on improving the code coverage.

Although DSmith is not coverage-oriented, we collect coverage information to help measure the test effect of DSmith. Software code coverage is closely related to software compile-time options, especially for large programs like compilers. In order to make a meaningful comparison, we choose the increment of 3 indicators for measurement, including the number of covered lines, the number of covered functions, and the number of covered branches. We use GCC-7 as the fuzzing targets.

We randomly select 200 test programs from the dataset. After that we compile them using GCC and collect compiler coverage as a baseline. Then DSmith uses these test programs as seed programs, and applies four generation strategies for one round of generation. During the generation process, every possible starting/insertion point is used to generate once, and truncation is performed at every possible truncation points to generate as many different test programs as possible. In addition to the baseline, we also compare the result of DSmith with the seq2seq model. We perform three repeated experiments using different seed programs, and predict the next tokens with greedy sampling. The average of the results is taken as the final result.

The results of the coverage improvement brought by test programs are shown in Fig. 8. The numbers of valid programs generated with each strategy are shown in Fig. 7. Overall, each of our generation strategies brings code coverage improvements. Since more valid test inputs are generated, DSmith is more efficient than the seq2seq method. When using strategy 1, although the seq2seq model generates more valid inputs, the code coverage improvements brought by these test programs are not as effective as those generated by DSmith. The number of valid test programs generated with strategy 2 is the largest of the four strategies, which ultimately leads to the largest increase in code coverage. It is worth noting that strategy 4 produces far fewer valid test cases than strategies 2 and 3, but it is efficient in improving code coverage due to the introduction of new function definitions.

TABLE I
REPORTED BUGS.

ID	Affected versions
bug-93160	7/8/9/10
bug-93072	7/8/9/10
bug-92799	7/8/9/10
bug-92725	7/8
bug-92615	7/8/9/10
bug-92478	8
bug-92469	7/8/9/10
bug-92377	7/8
bug-92368	7
bug-92355	7/8
bug-92352	7/8/9/10

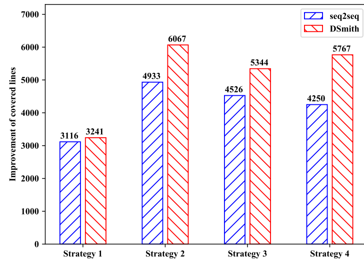
C. Bugs

We apply DSmith to the fuzz testing of GCC. Bugs in obsolete compiler versions are generally not interesting. We test three GCC versions that are still supported by the GCC development team. The three versions are GCC-7, GCC-8 and GCC-9. In addition, we test GCC-10.0, which is currently under development and will be released soon.

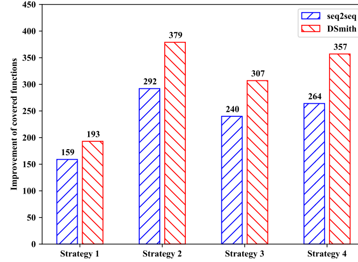
We have found 11 bugs, 6 of which even affected all the supported versions, including GCC-10.0. The bug numbers and the affected versions are shown in Table.I. All bugs information has been submitted to the GCC team. To our knowledge, this is the first time a bug has been found in the latest version of GCC using a deep neural networks-based generative model.

We list three test inputs generated by DSmith that successfully triggered bugs in GCC to further illustrate the usefulness of DSmith.

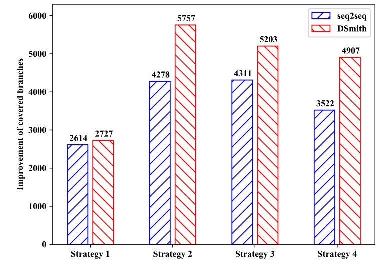
a) Bug-93160: This bug affected all GCC supported versions, including GCC-10.0. The test input was generated during source code completion process. Raw seed program does not trigger any bug. DSmith selected the beginning of line 3 as the starting point for the completion of this program, so lines 3 to 16 were discarded. DSmith generated lines 17 to 22, which then triggered a crash. This example demonstrates



(a) Improvement of Lines Numbers



(b) Improvement of Functions Numbers



(c) Improvement of Branches Numbers

Fig. 8. Improvements of Compiler Code Coverage

the effectiveness of the source code completion strategy in fuzz testing.

```
1     extern long unsigned int sub_0 ( const char * )
        ;
2     extern void sub_1 ( long unsigned int ) ;
3   - extern int sub_2 ( void ) ;
4   - void sub_3 ( int arg_0 ) {
5   -     char * var_0 = "STRING" ;
6   -     if ( arg_0 ) {
7   -         var_0 = "STRING" ;
8   -         goto label_0 ;
9   -     }
10  - label_1 :
11  -     sub_2 ( ) ;
12  - label_0 :
13  -     if ( sub_2 ( ) )
14  -         goto label_1 ;
15  -     sub_1 ( sub_0 ( var_0 ) ) ;
16  - }
17  + extern int var_0 ;
18  + void * var_1 = & var_0 ;
19  + register int var_0 asm ( "%ecx" ) ;
20  + char * sub_2 ( char * arg_0 ) {
21  +     return sub_1 ( sub_0 ( arg_0 ) + 1 ) ;
22  + }
```

b) Bug-92377: This test case was generated during lines of code insertion process. Raw seed program does not trigger any bug. After selecting line 9 as the insertion position, DSmith generated three lines of code and inserted them into that location. The added 3 lines of code caused a compiler segmentation fault. The bug affected GCC-7 and GCC-8. This case shows the effectiveness of the lines of code insertion strategy.

```
1     int sub_0 ( void ) ;
2
3     int __attribute__ ( ( returns_twice ) ) sub_1 (
        void ) ;
4
5     void sub_2 ( ){
6       int var_0 ;
7   +   if ( var_0 > 10 )
8   +     exit ( 0 ) ;
9   +   return 0 ;
10      var_0 = sub_0 ( ) + 2 + ( sub_1 ( ) + 1 +
       sub_1 ( ) ) ;
11    }
```

c) Bug-92469: This test case was generated during guided conditional branch insertion process. Raw seed program does not trigger any bug. First, DSmith selected line 6 as the insertion point. Then it added two tokens `if` and `(` at the end of the previous code, and passed them to the generative model. Following the two tokens `if` and `(`, a conditional statement was introduced. The addition of these two new lines eventually led to an internal compiler error. This example demonstrates the usefulness of guided branch insertion strategy for C compiler fuzzing. The shown program triggered an internal compiler error in GCC-9 and GCC-10.0. If we change "19" in line 3 to "20", it will also trigger a similar crash in older versions of GCC.

```
1     void sub_0 ( void ){
2       register int var_0 asm ( "19" ) ;
3
4   +   if ( var_0 )
5   +     return 0 ;
6     }
```

V. Related Work

a) Fuzzing techniques.: Fuzzing was introduced by Miller et al. [17] in 1990. Researchers have been using fuzzing as a standard method for software testing and bug finding. Fuzzing has been used to test UNIX utilities [17], compilers [1], [2], [18], runtime engines [19]–[22], and other kinds of applications. When fuzzing, fuzzers create a large number of inputs and run the target software with these inputs. In order to be effective, the generated inputs must be"valid enough" to bypass the early validation stage. Besides, the fuzzed inputs must be "invalid enough" to expose unexpected behaviors, such as an incorrect result, a freeze, or a crash.

For complex input formats, generation-based approaches prevent generated inputs from being rejected immediately by the target software. CSmith [2] generates random programs guided by a probabilistic grammar which covers a subset of the C programming language. It randomly selects an allowable rule from the grammar to generate C programs avoiding undefined and unspecified behaviors. LangFuzz [19] learns code fragments based on the given grammar. It generates new test programs by recombining fragments from a given test suite. LangFuzz has been successfully applied on JavaScript interpreter and PHP interpreter. However, several studies have

mentioned the difficulties of providing a fuzzer with syntax specifications for specific input formats [2], [22].

Various efforts have been invested to learn syntax from exsiting samples. To generate semantically-valid inputs, Skyfire [22] learns a probabilistic context-sensitive grammar (PCSG) from a large corpus of samples. The learned PCSG is then used to generate well-distributed seeds for fuzzing programs that process highly-structured inputs. This approach still requires providing a context-free grammar for the input format. TreeFuzz [20] uses learned probabilistic language models of structured data for generating test input data. TreeFuzz focus on input data that can be represented as a labeled, ordered tree. Although TreeFuzz does not require priori knowledge of the input format, complex model extractors have been designed to infer generative models from a given corpus of example data.

b) Deep Learning in fuzzing.: Deep learning has made significant breakthroughs in various fields of artificial intelligence. Advantages of deep learning include the ability to capture highly complicated features, weak involvement of human engineering, etc.

V-Fuzz [23] uses a neural network-based vulnerability prediction model to give a prior estimation on which parts of the software are more likely to be vulnerable. Then, a vulnerability-oriented evolutionary fuzzer is used to generate inputs which tend to arrive at these vulnerable locations.

Generation-based fuzzing is effective for fuzzing softwares with complex structured inputs. However, as we mentioned earlier, providing such grammars for fuzzers is a tedious task. Learn&Fuzz [21] use neural network-based statistical learning methods to learn input formats from a number of sample inputs. Godefroid et al. presented and evaluated the seq2seq model to automatically learn a generative model of PDF objects. Cummins et al. presented DeepSmith [18] for OpenCL compiler fuzzing. They used LSTM to model the programs. Liu et al. [1] proposed DeepFuzz, a fuzzing tool for C compilers based on a a character-level generative seq2seq model.

VI. Conclusion

In this paper, we design and implement DSmith to capture the long-distance syntax dependencies in programs for a robust compiler fuzzing inputs generation. We apply token-level neural language model of C programs to fuzzing. Attention mechanism is used to enhance the capabilities of the neural generative model to capture the long-distance syntax dependencies. According to the four novel generation strategies, DSmith can generate various effective test programs by its built language model.

Experiments show that DSmith increases the parsing pass rate of the generated programs by an average of 19% and significantly improves the code coverage of the compiler, compared with existing deep neural networks-based fuzzing methods. In our experiments, DSmith has found eleven brand new bugs in GCC compiler.

References

[1] X. Liu, X. Li, R. Prajapati, and D. Wu, “Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing,” in *Proceedings of the... AAAI Conference on Artificial Intelligence*, 2019.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[3] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] I. Sutskever, O. Vinyals, and Q. Le, “Sequence to sequence learning with neural networks,” *Advances in NIPS*, 2014.

[5] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, “Exploring the limits of language modeling,” *arXiv preprint arXiv:1602.02410*, 2016.

[6] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” *arXiv preprint arXiv:1506.00019*, 2015.

[7] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

[8] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint arXiv:1409.0473*, 2014.

[9] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, “Show, attend and tell: Neural image caption generation with visual attention,” in *International conference on machine learning*, 2015, pp. 2048–2057.

[10] A. M. Rush, S. Chopra, and J. Weston, “A neural attention model for abstractive sentence summarization,” *arXiv preprint arXiv:1509.00685*, 2015.

[11] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, “Hierarchical attention networks for document classification,” in *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, 2016, pp. 1480–1489.

[12] M.-T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” *arXiv preprint arXiv:1508.04025*, 2015.

[13] X. Yu, N. T. Vu, and J. Kuhn, “Learning the dyck language with attention-based seq2seq models,” in *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, 2019, pp. 138–146.

[14] M. Zalewski, “american fuzzy lop,” 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[15] Google, “Honggfuzz,” 2016. [Online]. Available: https://github.com/google/honggfuzz

[16] LLVM, “libfuzzer:a library for coverage-guided fuzz testing,” 2017. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[17] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[18] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, “Compiler fuzzing through deep learning,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 95–105.

[19] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 445–458.

[20] J. Patra and M. Pradel, “Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data,” *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[21] P. Godefroid, H. Peleg, and R. Singh, “Learn&fuzz: Machine learning for input fuzzing,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[22] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[23] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, “V-fuzz: Vulnerability-oriented evolutionary fuzzing,” *arXiv preprint arXiv:1901.01142*, 2019.