The Pennsylvania State University
The Graduate School

NEURAL PROGRAM SYNTHESIS FOR COMPILER FUZZING

A Dissertation in
College of Information Sciences and Technology
by
Xiao Liu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2019

The dissertation of Xiao Liu was reviewed and approved* by the following:

Dinghao Wu
Associate Professor of Information Sciences and Technology
The Pennsylvania State University
Thesis Advisor
Chair of Committee

Mary Beth Rosson
Professor of Information Sciences and Technology
The Pennsylvania State University

C. Lee Giles
Professor of Information Sciences and Technology
The Pennsylvania State University

Danfeng Zhang
Assistant Professor of Computer Science and Engineering
The Pennsylvania State University

*Signatures are on file in the Graduate School.

Abstract

Compilers are among the most fundamental programming tools for building software. However, production compilers remain buggy. GNU compiler collection (GCC), as a long-lasting software released in 1987, provided as a standard compiler for most Unix-like operating systems, has caught over 3,410 bugs from the day they were created. Fuzzing is often leveraged for stress testing purposes with newly-generated, or mutated inputs to find new security vulnerabilities. In our study, we propose a grammar-based compiler fuzzing framework called DeepFuzz that continuously synthesizes well-formed C programs to trigger internal compiler errors or "bugs", as they are commonly called. In this framework, we are interested in how to apply generative deep neural networks (DNNs), such as the sequence-to-sequence model, to synthesize well-formed C programs based on training through syntax-correct programs. We are also interested in how to synthesize programs using a novel form of reinforcement learning, where the model becomes its teacher to start with a random neural network with no training data and trains itself through self-play. We will use a synthesized set of new C programs to fuzz off-the-shelf C compilers, e.g., GCC and Clang/LLVM. This thesis describes our analysis of neural program synthesis for compiler fuzzing in three steps.

First, we conduct a first-step study by implementing DeepFuzz that deploys a sequence-to-sequence model to synthesize C programs. We have performed a detailed case study on analyzing the pass rate of generating well-formed programs and achieving the goal of fuzz testing, which requires a certain degree of variation. In general, DeepFuzz generated 82.63% syntax valid programs and improved the testing efficacy with regards to line, function, and branch coverage. It identified previously *unknown* bugs, and 8 of them were confirmed by the GCC developers.

Second, for the cases when we could not get any or enough data to train a model for representing the grammar, we build a reinforcement learning framework for program synthesis and apply it to the BF programming language. With no training data set required, the model is initialized with random weights at the very beginning, and it evolves with environment rewards provided by the target compiler being tested. During the performance of the learning iterations, the neural network model

gradually learns how to construct valid and diverse programs to improve testing efficacies under four different reward functions that we defined. We implemented the proposed method into a prototyping tool called AlphaProg. We performed an in-depth diversity analysis of the generated programs that explains the improved testing coverage of a target compiler being tested. We reported two important bugs for this production compiler and they were confirmed and addressed by the project owner.

Third, we extend the framework to synthesize C programs, which is more challenging in terms of state space. We propose an automatic code mutation framework called FuzzBoost that is based on deep reinforcement learning. By adopting testing coverage information collected at runtime as the reward, the fuzzing agent learns to fuzz a seed program that achieves an overall goal of testing coverage improvement. We implemented this new approach, and preliminary evidence showed that reinforcement fuzzing can outperform baseline random fuzzing on production compilers. It also showed that a pre-trained model can boost the fuzzing process for seed programs with similar patterns.

This thesis solves the problem of using the DNN to synthesize new programs for compiler fuzz testing. Specifically, the proposed framework is able to handle compilers of different programming languages. Accordingly, DeepFuzz and FuzzBoost are designed for the C compiler testing, and AlphaProg is designed for the BF language compiler testing. Additionally, the generative neural networks for program synthesis can be trained with or without training data. Moreover, the model in DeepFuzz is trained based on training data but AlphaProg and FuzzBoost rely on reinforcement learning, which requires no training samples. We built prototyping tools for each study and applied them for practical use. Their effectiveness was evaluated, and they caught real bugs in off-the-shelf compilers.

Table of Contents

List of Figures

List of Tables

Acknowledgments

On drafting this thesis, I have come across many pebbles along the way and finally end up with this milestone. Foremost, I would like to express my sincere gratitude to my advisor, Prof. Dinghao Wu, without his help, I cannot make even a small step.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. C. Lee Giles, Prof. Danfeng Zhang, and Prof. Mary Beth Rosson, for their encouragement, insightful comments. Especially for Prof. Mary Beth Rosson, who is on the committee for my master thesis, my PhD candidacy exam, and my PhD thesis. She is always very supportive and responsive for handling my questions, concerns, and requests.

I would also give thanks to my fellow labmates especially Dr. Shuai Wang, Dr. Yufei Jiang, Dr. Pei Wang, Xiaoting Li (PhD Student), Rupesh prajapati (PhD Student), for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last six years.

Last but not the least, I would like to thank my family, especially my husband Dr. Feng Sun, that provides me through moral and emotional support in my life. My sincere thanks also goes to my parents Aiqin Tan and Jingtao Liu for giving birth to me at the first place.

The dissertation is based on research supported in part by the National Science Foundation (NSF) under grants CNS-1223710, CCF-1320605, and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894. We also gratefully acknowledge the support of NVIDIA Corporation with the donation of two Titan Xp GPUs used for this research. Findings and conclusions in this dissertation do not necessarily reflect the view of the funding agencies.

Chapter 1
Introduction

1.1 Motivation

Compilers are among the most fundamental components of computation systems, and they are important parts of the trusted computing base of our machines. However, they often contain a substantial number of bugs. GNU compiler collection (GCC) is a long-lasting software released in 1987 that was developed to provide a standard compiler for most Unix-based operating systems, and it caught over 3,410 internal bugs [98] from the day they were created. Even in a recent research paper, 217 unknown bugs detected among which 119 were fixed [103] over a six-month study period. Similar circumstances apply to Java, Python, and JavaScript, with thousands of bugs being detected in widely used compilers, among others. Compiler bugs can result in unintended program executions and lead to catastrophic consequences in security-sensitive applications, and they may also hamper developers productivity in debugging a program when the root cause cannot be identified in the applications or compilers. Consequently, improving compiler accuracy is important, although validating compilers is not easy, as the size of code bases continues to gradually increase. The code base of today's GCC includes approximately 15 million lines of code [84], which is close to the 19 millions of lines of code.

It is critical to make compilers dependable, and in the past decade, compiler

verification has become a paramount concern, which has led to verification grant challenges in computing research [33]. Mainstream research focuses on formal verification [49, 51], translation validation [73, 76], and random testing [46, 47, 48, 53]. The objective of the first two categories is to provide certified compilers, e.g., CompCert [50], which represents promising progress in this area, though in practice, it is challenging to apply formal techniques to fully verify a production compiler because it requires more effort to compose the specifications than build a compiler. Therefore, testing remains the dominant approach to conducting compiler validation.

The focus of this study was on compiler testing. By scanning programs covering different features to different production compilers that turn on different levels of optimization, internal compiler errors (i.e., genuine bugs in the compiler) may be triggered during the compilation with a detailed error message indicating what and where the error is. However, challenging hurdles in compiler testing include determining how to generate "good" programs to make testing more efficient, how to automate this process, and how to build a continuous testing framework. Existing methods including man-made tests, each of which covers some features, and it is common today to see the gradually enlarged test suites for modern compilers. Man-made test suites are efficient for testing in terms of coverage, though it takes huge human effort to develop these tests. Nevertheless, a practical way to reduce human labor for testing is fuzz testing (i.e., "fuzzing"). Fuzzing [7] is the process of finding security vulnerabilities by repeatedly executing a program with automatically generated/modified inputs and detecting abnormal behaviors by observing the execution results. The primary techniques for input fuzzing in use today are black-box random fuzzing [101], white box constraint-based fuzzing [27], and grammar-based fuzzing [22]. Black box and white box fuzzing are fully automatic and have historically proven effective at finding security vulnerabilities in binary-format file parsers. In

contrast, grammar-based fuzzing is required an input grammar specifying the input format of the application under test, which is typically written by hand. This process is labor-intensive, time-consuming, and error-prone. Nevertheless, grammar-based fuzzing is the most effective fuzzing technique known today for fuzzing applications with complex structured input formats, e.g., compilers. For compiler testing, one way to deploy the grammar-based fuzzing is to encode the C grammar as rules for test case generation. However, in practice, C11 [17], the current standard of C syntax, has 696 pages of detailed specifications, which represents a hurdle for engineers when constructing such a grammar-based engine.

1.2 Problem Definition

In this thesis, we consider the problem of automatically generating syntactically valid inputs for grammar-based fuzzing with a deep learning framework. More specifically, we target on training a generative deep neural network which can be viewed as an implicit representation of the "grammar", to be more precise, the language patterns, for programming languages. In this thesis, we consider two scenarios, one with and one without training data set.

The two scenarios describe the two main challenges incrementally in grammar-based fuzz testing. To test general compilers that are well maintained, i.e. GCC, that we can find enough test programs accordingly, we aim at building an end-to-end learning framework to continuously generate valid programs by encoding programming language patterns in neural networks. It reduces human effort in constructing rule-based production rules in previous grammar-based fuzzing engines. However, for less well-maintained compilers, even parsers or interpreters that we cannot find enough data for training such a model to represent language patterns, we aim to build a reinforcement learning framework to achieve the same outcome.

Simply stated, a deep neural network will be incrementally trained with a reward system. The second scenario applies to a more general case, that is, any software systems that take in highly-structured inputs and can provide an environmental reward to describe the validity of given inputs.

To train a deep neural network from a training data set, we propose to apply existing generative models, such as the *Sequence-to-Sequence* model [86] in a supervised learning strategy leveraging the original test suites provided with production compilers. At a high level, the sequence-to-sequence model consists of two recurrent neural networks: the encoder RNN simply consumes the input source code without making any prediction; the decoder, on the other hand, processes the target sentence while predicting the next words. Originally, it was widely used for machine translation [42], which takes in a sequence of the original language and generates the sequence on a target language with the same semantics. It has also been applied to text generation [85], where feeding in abundant paragraphs of Harry Potter results in the automatic generation of a new paragraph that keeps the style of writing of what J.K. Rowling wrote. Theoretically speaking, by training the model on the original paragraphs, we implicitly encoded the correct spelling of words, valid syntaxes of sentences, and detailed styles of writing behaviors into a generative model. The same idea can be applied to program synthesis, where it is only necessary 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, which includes more than 10,000 short programs that cover most of the features specified in the C11 standard. Moreover, the open-source projects, online coding systems, students programming assignments are all good sources for grabbing syntax-valid programs for training.

1.3 Research Objective

In general, we used neural program synthesis to accomplish two main objectives. The first is to generate new programs that follow legitimate grammar with and without a set of syntactically correct programs. The major challenge comes from handling long sequences and grammar representation. The second objective was to improve compiler testing efficacy. We targeted improving the coverage and capture of more internal errors in production compilers. More specifically, we wanted to enhance line coverage, function coverage, and branch coverage for testing production compilers. The coverage information is important for compiler testing as it indicates how many more lines of code are covered by executing a target program with a set of new inputs. Intuitively, the more lines of code that are executed, the more chances there are to assure a software is secure in terms of program semantics. Also, during the fuzzing process, we sought to detect unknown compiler bugs with the improved test suites. There are two stages in the entire workflow, program generation and compiler testing. We targeted production compilers, such as GCC [26] and LLVM/Clang [15].

We conducted some preliminary studies. We have pipelined a prototype with around 5,000 pieces (20 MB) of valid C programs collected from the GCC repository and online coding systems. We trained an LSTM model with 2 layers and 512 hidden units per layer. We trained the model for in total 30 epochs on a server machine with 2.90 GHz Intel Xeon(R) E5-2690 CPU and 128 GB of memory. It takes 900 seconds for an epoch and 7.5 hours for the entire training. We generated a total of 4,409 programs by inserting two new lines of code into randomly sampled seed programs. Among the newly generated programs, 1,134 of them are syntax valid C programs which mean the generation success rate was about 25.72%. In the generated syntax invalid programs, we observed some common errors such as "undeclared identifier" (2,509) which indicates that some variables are used before

they are defined, and also "expected expression" (1,823), which includes syntax errors like unbalanced parenthesis exist. We want to improve the model to enhance the generation success rate in our next step, though our machine is not powerful enough to handle (1) more training data being feed in; (2) building of self-training cycles based on reinforcement learning as state-of-the-art approaches. We would also like to pipeline the current prototype with coverage analysis. More specifically, we will analyze how is the percentage of code, paths, and branches coverages with the newly generated programs leveraging static analysis. We will be interested in answering the following research question: How can syntax-valid programs be automatically generated for fuzzing compilers and will these newly generated programs efficiently improve the testing coverage and detect unknown bugs?

In the first stage, we will train generative neural networks based on conventional deep learning models, i.e. the sequence-to-sequence model, from a set of training data; or based on a reinforcement learning framework, such as the AlphaGo Zero, which trains itself through data and reward from a compilation environment. After training the model within certain epochs, we started to generate new programs with this fitted model. For program generation, we tried different generation strategies, such as direct insert and replace. Because our target was to fuzz production compilers, we aimed to generate programs that cover the most features of the C language or BF language. Therefore, we also adopted some sampling methods to diversify the generated program.

In the second stage, we fed the generated C programs or BF programs, either syntactically correct or incorrect, to compilers in different optimization levels and log the compiling messages. The message is a flag for (1) whether a generated program is syntactically correct or (2) the generated program may have triggered an internal compiler error (bug) of the specific compiler at a specific optimization level. In

addition to the compiling message, we logged the execution trace to provide coverage information. In a nutshell, for this program generation task, we had three objectives: to generate syntax valid programs, to improve code coverages, and to detect new bugs. We performed deliberate studies on three related metrics, pass rate, coverage, and bugs, for the three objectives.

1.4 Research Question

Our work is the first that works on grammar-based compiler fuzzing with a deep learning framework. We have three main research questions:

- Can we build a continuous testing framework that automatically generates syntactically correct programs based on deep neural networks for compiler fuzzing based on observing existing syntactically correct programs?
- If there is no training data to rely on, how can a deep neural network be trained to generate programs for compiler fuzzing based on reinforcement learning?
- What are the key factors how these key factors will affect the accuracy of the generative model and fuzzing performance. How testing coverage (line, function, branch) is increased with our proposed method?

To ansser these research questions, we conducted three research projects. To answer the first question, we developed DeepFuzz to train a generative deep neural network over training data of well-formed C program and use the trained model to continuously produce new C programs. To answer the second and third question, we conduct two studies: we developed AlphaProg and FuzzBoost. AlphaProg is a generative fuzzing framework that trains itself a deep neural network to producing BF language programs at the character-level. It uses environment rewards from compilation information from off-the-shelf compilers, i.e. BFC; FuzzBoost is

a mutational fuzzing framework based on a deep reinforcement learning system as well. It predicts actions to choose from pre-defined mutation rules to mutate seed C programs by adopting testing coverage information collected from runtime information as the reward. The three studies helped us to better construct a neural program synthesis framework for compiler fuzzing.

The rest of this thesis is organized as follows.

- We first review the related works about Compiler Testing, ML for Program Synthesis, and Advanced ML for Program Synthesis in Chapter 2.

- We then report our study of DeepFuzz that uses a Sequence-to-Sequence model to synthesize program for compiler fuzzing in Chapter 3. We present a detailed case study on analyzing the pass rate of generating well-formed programs and achieving the goal of fuzz testing, which requires a certain degree of variation in the synthesized new programs. We analyzed the performance of DeepFuzz with 3 types of sampling methods, as well as 3 types of generation strategies. Consequently, DeepFuzz improved the testing efficacy with respect to line, function, and branch coverage. In our preliminary study, we found and reported 8 bugs in GCC, all of which have been actively addressed by developers.

- Next, to solve the case when there are no/few training data, we report our study AlphaProg in Chapter 4. It is a generative fuzzing framework based on reinforcement learning. A naive model was first provided and it evolves with the rewards provided by the target compiler we are going to test. By iterating the learning cycle, the model learns how to write valid programs and generate programs that improve the testing efficacy. We analyzed the framework with 3 different reward functions, and our study revealed the effectiveness of

AlphaProg for compiler testing. We also performed an in-depth diversity analysis of the generated programs, which explained the improved testing coverage of our target compiler. We reported two important bugs for this production compiler, and they were confirmed and well-addressed by the project owner.

- In addition, we extended the reinforcement learning framework into FuzzBoost to continuously produce C programs for fuzzing GCC in Chapter 5. It is a mutational fuzzing framework. By adopting testing coverage information collected from runtime information as the reward, we developed a learning system with the state-of-the-art deep Q-learning algorithm that optimizes this reward. In this way, the fuzzing agent learns the actions to perform to fuzz a seed program that achieves an overall goal of testing coverage improvement. We have implemented this new approach and evidence showed that reinforcement fuzzing can outperform baseline random fuzzing on production compilers. It also showed that a pre-trained model can boost the fuzzing process for seed programs with similar patterns.

- Then, similarities and differences among the three projects are discussed in Chapter 6. We compare DeepFuzz and FuzzBoost in terms of the valid rate of synthesized programs, since they are both fuzzers of C compilers; we cannot compare the two in terms of coverage because FuzzBoost focuses on single seed mutations. We also discuss the models we chose in the three projects, where we use encoder-decoder (RNN) for program generation as in DeepFuzz and AlphaProg; and RNN embedding plus DNN for program token-based mutations in FuzzBoost.

- Finally, conclusions are presented in Chapter 7.

Chapter 2
Related Work

Our research seeks to detect more bugs in production compilers and improve the testing coverage in the meantime. In this section, we review related works on compiler testing, program synthesis and also discuss existing methods in program-related usage of deep learning models.

2.1 Compiler Testing

Compilers are one of the most fundamental components of any computing system, but studies have shown they to be buggy [84]. To assure the correctness of a compiler, researchers have proposed different methods, most of which focus on verification and testing.

In the past decade, compiler verification is an active area and has been discussed widely [33]. Mainstream research focuses on formal verification [49, 51], and translation verification [73, 76]. The proposed method provides a fully certified compiler, e.g., CompCert [50] which is proven to be correctly functioned. However, it does not apply to production compilers, the code base of which is exceptionally large. It takes more effort to compose the specifications for verifying such systems than building a new one. Therefore, using testing techniques remains the dominant approach in compiler validation [46, 47, 48, 53]. Existing methods for compiler testing include

man-made tests that cover most features of one syntax. However, it is laborious and a practical way to reduce human labor is fuzz testing [7], which is the process of finding security vulnerabilities by repeatedly executing a program with newly generated inputs. There are three major methods to fuzz inputs: black box random fuzzing [9, 101, 103], white box constraint-based fuzzing [27], and grammar-based fuzzing [25]. The first two methods are fully automated, while grammar-based fuzzing requires human involvement, where grammar is composed according to rules.

Grammar-based fuzzing [22] uses an existing corpus of language grammars for fuzzing. Grammar-based fuzzing requires an input grammar to specify the input format of an application being tested. Despite the high effectiveness of grammar-based fuzzing to synthesize complexly structured inputs, these grammars are typically written by hand [98], which makes the process laborious, time-consuming, and error-prone. 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. However, in practice, C11 [17], the current standard of the C programming language, has 696 pages of detailed specifications, which represents a hurdle for engineers when constructing such a grammar-based engine. A few automation methods have been proposed for grammar-based fuzzing to save human labor, including statistic-based [91], mutation-based [39], and deep learning-based [18, 60]. Researchers utilized an RNN-based model to encode program grammar and generate new well-formed C programs for compiler fuzzing. In this paper, we describe how our method boosts the generation process by using a deep neural network to predict the generation based on an observation of self-generated programs and corresponding rewards from the environment. This method makes the compiler testing work for cases where few training data can be acquired, such as the BF language.

Mutation-based fuzzing uses an existing corpus of seed inputs for fuzzing. It

generates new inputs by modifying the provided seeds. A well-known fuzzer that is mutation-based is called AFL [101], which randomly mutates seed inputs and incrementally adds new seeds to the set concerning defined heuristics. Several boosting techniques have been proposed to improve the efficiency of mutation-based fuzzing. AFLFast [8] boosts original AFL fuzzer by focusing on low-frequency paths that allow the fuzzer to explore more paths with limited time. Skyfire [91] applies grammar in existing seed inputs for fuzzing programs that take highly-structured inputs. Kargen and Shahmehri [39] perform mutations on the machine code instead of a well-formed input, which they can use the information about the input format encoded in the generated program to produce high-coverage inputs. DeepFuzz [60] utilized an RNN-based model to encode program grammar and generate new well-formed C programs for compiler fuzzing. In this paper, our method boosts the mutation process by using a deep neural network to predict the mutation based on an observation of existing seed programs.

2.2 Program Synthesis

Program synthesis is one of the fundamental problems in artificial intelligence (AI) which aims at synthesizing programs automatically that follow certain predefined specifications. It can be traced back to Waldinger and Lee [90], where a theorem prover was used to construct LISP programs based on a formal specification of the input-output relation. Since formal specifications are often as complex as writing the original program, many researchers have proposed different techniques to achieve the same goal with simpler partial specifications in the form of input-output examples [4, 83]. Rule-based synthesis approaches have been successful in pushing the process further [54, 57, 58, 59, 62], which used rule-based systems to translate user specifications in natural language into program commands. Meanwhile, DSL-based

inductive synthesis has also been another milestone, with the most widely known example being the FlashFill system in MS Excel [31]. However, such systems are difficult to extend and need significant development from domain experts to provide the pruning rules for supporting more efficient search. As a result, the use of machine learning methods have been proposed based on Bayesian probabilistic models [52]. Researchers also proposed the inductive logic programming [71] to automatically generate programs based on examples. In recent studies, inspired by the the success of Neural Networks in other applications, differentiable controllers were made to learn the behavior of programs by using gradient descent over a differentiable version of traditional programming concepts such as memory addressing [29], manipulating stacks [30, 38], and register machines [44]. However, their method of solving problems is still not scalable because they learn a different model for each program, but which is later tackled in Bunel's work [10], where a single model is used for learning a large number of programs.

Incorporating knowledge of target domain's grammars to enforce syntactical correctness has already proven useful to model arithmetic expressions, molecules [45], and programs [75, 99]. These approaches define the model over the production rules of the grammar. In our scenario, the only specification of the synthesized program is syntactically correct. This specification is much looser than any previous program synthesis problems in which the specifications are usually at the semantic level. However, we have more challenges regarding how to generate more diverse programs to cover the features in compilers and therefore increase the testing efficacy. Besides, our framework will serve as a syntax guard for any program synthesis research where manually constructed production rules can be replaced with an automatically trained neural network.

2.3 Program-related Advanced Machine Learning

Reinforcement learning is adopted in many sequential action prediction tasks after its first success in the game of Go [82], and thus as the task of program synthesis. Based on given specifications, Bunel et al. performed reinforcement learning on top of a supervised model with an objective that explicitly maximizes the likelihood of generating semantically correct programs [10]. There are also research projects that target program semantics, such as Neurally Directed Program Search (NDPS) [89], proposed for solving the challenging non-smooth optimization problem of finding a programmatic policy with maximal reward. Overall, existing projects that adopt deep reinforcement learning for semantic program synthesis focus on one semantic goal with one synthesis task. Our objective is to generate source programs that are well-formed but contain different syntactic features, which are similar to the work from Bottinger et al. [9] that aims at PDF parser fuzzing. In our design, we consider the improvement of testing coverage of compilers as the reward for reinforcement learning.

AI-based software security and software analysis have been discussed over the years [23, 72, 102]. Neural network-based models dominate a variety of applications, and interest has grown tremendously regarding their use for program analysis [2, 74] and synthesis [21, 55]. 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, auto-completion, and repairing syntax errors [6, 32]. 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 [18, 28] which learns a generative model to produce PDF files to fuzz the PDF parser.

Chapter 3
Program Synthesis based on S2S for Compiler Fuzzing

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 to find new bugs or security vulnerabilities. In this study, we proposed a grammar-based fuzzing tool called DeepFuzz. Based on a generative *Sequence-to-Sequence* model, which can be viewed as an implicit representation of the language patterns for training data, 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 with respect to the line, function, and branch coverage. We apply our DeepFuzz technique to test GCC and Clang/LLVM. During our preliminary analysis, we found and reported 8 bugs in GCC, all of which have been actively being addressed by developers.

3.1 Problem

Compilers are among the most fundamental components of computation systems, and they are part of the trusted computing base of our machine. But they contain numerous bugs. GCC, as a long-lasting software released in 1987, provided a standard compiler for most Unix-like operating systems, caught over 3,410 internal bugs [98] from the day they were created. Even in a recent research study, 217 unknown bugs were detected, among which 119 were fixed [103] over a six-month study. Similar situations apply to Java, Python and JavaScript, over thousands of bugs are detected in widely used compilers, let alone the others. Compiler bugs can result in unintended program executions and lead to catastrophic consequences in security-sensitive applications. It may also hamper developersâĂŹ productivity in debugging a piece of program when the root cause cannot be identified in the applications or compilers. Therefore, improving compiler correctness is important. However, validating compilers is not easy with the gradually enlarged code base: the code base of todayâĂŹs GCC includes around 15 million of lines of code [84], which is 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 [33]. Mainstream research focuses on formal verification [49, 51], translation validation [73, 76] and random testing [46, 47, 48, 53]. The first two categories try to provide certified compilers, e.g., CompCert [50], which made promising progress in this area. However, in practice, it is challenging to apply formal techniques to fully verify a production compiler because it requires more effort to compose the specifications than to build a compiler. Therefore, testing remains the dominant approach in compiler validation.

Our work focuses on compiler testing. By loading 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, which results in a detailed error message indicating what and where the error is. However, it is challenging to generate "good" programs to make testing more efficient and build a continuous testing framework by automating this process. Each test, including man-made ones, and 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 [7] is a method used 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 [101], white box constraint-based fuzzing [27], and grammar-based fuzzing [22]. 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. By contrast, grammar-based fuzzing requires input grammar specifying the input format of the application being tested, 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. However, in practice, C11, the current standard of the C programming language, has 696 pages of detailed specifications, which represents a hurdle for engineers when constructing such a grammar-based engine.

In this study, we considered the problem of automatically generating syntactically valid inputs for grammar-based fuzzing with a generative recurrent neural network. More specifically, we targeted training a generative neural network, which is an implicit representation of "grammar", or to be more precise, the language patterns, for input data. We proposed to train a Sequence-to-Sequence model in a supervised learning strategy leveraging the original test suites provided with production compilers. At a high level, the Sequence-to-Sequence model consists of two recurrent neural networks: the encoder RNN simply consumes the input source code without making any prediction, while the decoder processes the target sentence and predicts the next words. Originally, it was widely used for machine translation, which takes in a sequence of the original language and generate the sequence in a target language with the same semantics. It has also been applied to text generation. where feeding in abundant paragraphs of Harry Potter results in the automatic generation of a new paragraph that maintains the style of J.K. Rowling. Theoretically speaking, by training the model on the original paragraphs, we implicitly encode the correct spelling of words, valid syntaxes of sentences, and 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, which has more than 10,000 short programs that cover most of the features specified in the C11 standard. In addition, the open-source projects, online coding systems, students programming assignments are all good sources for grabbing syntax-valid programs for training. On the training stage, we tune the parameters in the neural network to encode the language patterns for C programs into the model, and based on this, we will 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 training data, C syntax in our scenario, our framework DeepFuzz will continuously provide new well-formed C programs.
- Second, we build a practical tool for fuzz testing off-the-shelf C compilers. We conduct a detailed analysis regarding how key factors will affect the accuracy of the generative model and fuzzing performance. The testing coverage (line, function, branch) is increased with our proposed method.
- Third, we apply our DeepFuzz technique to test GCC and Clang/LLVM. During our preliminary analysis, we found and reported 8 (will increase later) real-world bugs. These bugs have been actively addressed by developers.

3.2 Overview

3.2.1 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

$$h_{\langle t \rangle} = f(h_{\langle t-1 \rangle}, x_t) \tag{3.1}$$

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

$$p(x_{t,j} = 1|x_{t-1}, ..., x_1) = \frac{\exp(w_j h_{\langle t \rangle})}{\sum_{j=1}^{K} \exp(w_j h_{\langle t \rangle})}, \tag{3.2}$$

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

$$p(x) = \prod_{t=1}^{T} p(x_t|x_{t-1}, ..., x_1). \tag{3.3}$$

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. [13] 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

$$h_{\langle t \rangle} = f(h_{\langle t-1 \rangle}, y_{t-1}, c), \tag{3.4}$$

and similarly, the condition distribution of the next character is

$$p(y_t|y_{t-1}, ...y_1, c) = g(h_{\langle t \rangle}, y_{t-1}, c), \tag{3.5}$$

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 [14]. 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 [28]. 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 [81] 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 study, 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.

3.2.2 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 3.1 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 [26] and LLVM/Clang [15]. 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.

On the second stage, we feed the generated C programs, either syntactically correct or incorrect, to the compilers in different optimization levels. In addition to the compiling message, we log the execution trace to provide the coverage information. 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*.

[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 [16]. 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.*

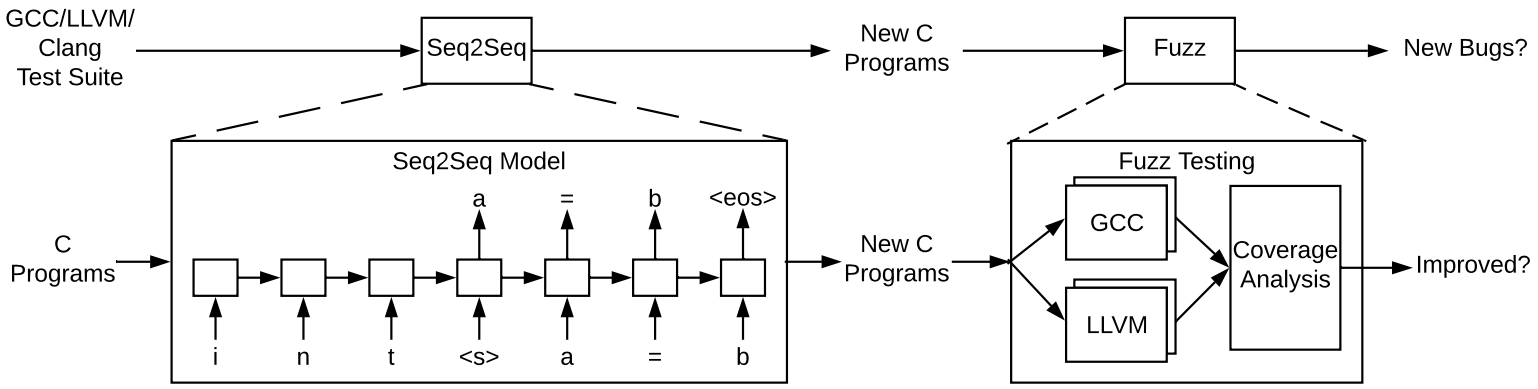


Figure 3.1. Workflow of DeepFuzz

3.3 Design

Before we set up the training stage, we first concatenate all the C programs in the training set into a single file. By connecting these files, we have a large sequence of characters. We then 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 are 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 The comments are usually described in natural language which do not follow the syntax of C programming language. Therefore, they are noise to us. We first cut off 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. Observing the training dataset, we see white spaces are not unified formated. For example, in some programs, there is a white space before and after the operator but in the others, there is no such pattern; and in some programs, programmers use a tab for indentation but in the others, the indentation is marked by four or two spaces. To unify program style, we replaced all the white space characters with a single space. In addition, we delete all the spaces before and after operators to make a full expression more condensed.

Macro Macro is a unique feature of the C programming language. A macro is a fragment of code which has been given a new name. Whenever the name is used, it is always replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. Object-like macros resemble

data objects when used, function-like macros resemble function calls. However, the use of macro will cause problem for the training process as it brings noise. For example, if we define a macro #define OnePlus 1+, we can write a = OnePlus 1 later in the program but which does not follow C syntax. To avoid this situation, we replace all the macro names with the contents as defined in the preprocessing.

3.3.1 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. To be more specific, based on the predicted possibility distribution of a next character, we always pick the one with the highest possibility. This method will generate programs that are most likely to be well-formed and consistent, but it will limits the number of different programs that can be generated.

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. This sampling method can help with generating a more diverse set of programs as it can combine different patterns it learnt from the training dataset. However, it will trigger the low pass rate problem, in another word, it is highly possible that a newly generated program is syntactically invalid. To balance the diversity and

well-formness, we set up a threshold in the sampling. That is to say, every time we only sample among the most predicted characters.

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. We propose this method because, we hope to predict a next character more consistent within the prefix token but there can be more freedom when predicting the starting character of next token. This method is expected to generate more well-formed C programs compared with the *Sample* method and enhance the diversity compared with the *NoSample* method.

3.3.2 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 (We use lines of code instead of C syntactic objects such as statements since we treat C programs purely as sequences of characters.) 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.

Table 3.1. Model configuration

	Training Size	# of Layer	# of Hidden Unit
Text Generation[2]	100 MB	1 (*RNN*)	1,500
Learn&Fuzz[3]	534 PDF Files	2 (*LSTM*)	128
DeepFuzz	10,000 C Programs	2 (*LSTM*)	512

3.4 Evaluation

3.4.1 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 compare our configuration with others in Table 3.1. In a previous study on text generation [85], 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 [28], 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, 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 are covered by the tests, the more certainty we assure the completeness of testing. There are three kinds of coverage information we collect during our analysis:

line coverage, function coverage, and branch coverage. Line coverage identifies how many lines were encountered as a result of your tests; function coverage identifies how many functions are covered by your test; and branch coverage identifies how many branches in your program are covered. 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 compilers in different optimization levels, it is expected to trigger bugs like crashes or wrong 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 a normal parsing process and the error message will help us to find bugs in compilers.

Post-processing: We tested a trial generation, where 4,409 program were generated with no sampling applied and using the generation strategy *G1*, which is to insert two new lines at one location of original seed programs. We analyzed results from this trial: Among the newly generated programs, 1,134 of them are syntax valid C programs which means the pass rate is only about 25.72%. To increase the pass rate, we took a closer look at the generated programs. In the generated syntax invalid programs, we observed a common error called Undeclared Identifier which indicates that some identifiers are used before they are declared. There are 2,509 programs are syntax invalid due to this problem. Therefore, we post-process the generated program to handle this issue. We used a try-catch for this error and automatically declare the undeclared identifiers at the beginning of this program. Although this issue is handled in an ad-hoc way, but by adding this post-processing process, the pass rate will be remarkably enhanced.

3.4.2 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 3.2 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*.

Result: Figure 3.2 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.

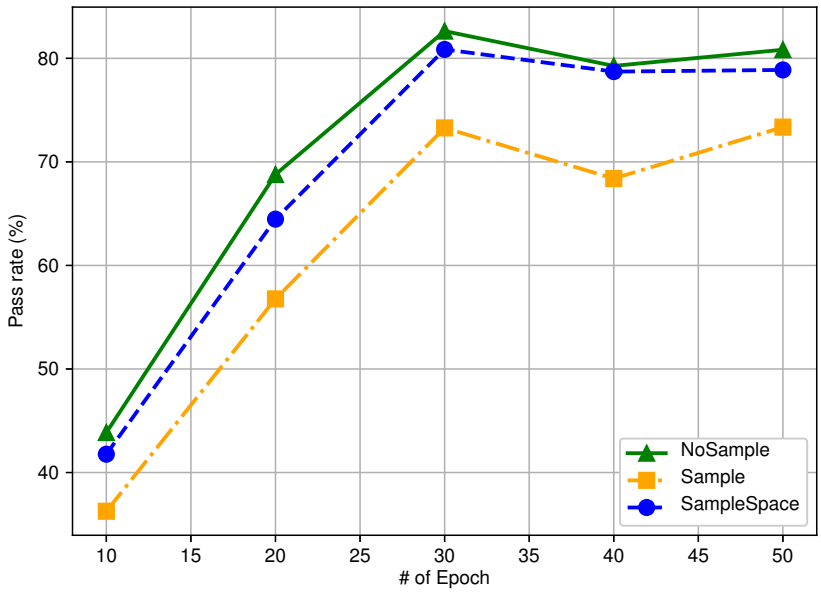


Figure 3.2. Pass rate for different sampling methods

- 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.

Result: Table 3.2 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

Table 3.2. Pass rate of 10,000 generated programs

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

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.

3.4.3 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/branches are covered with the original seed test suite (10,000) plus the newly generated test suite (10,000) specifically for GCC-5 and Clang-3. In addition, to analyze how sampling methods can influence the coverage improvements, we record the coverage improvement percentages under different sampling methods. We recorded the coverage improvement information in Table 3.3 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 3.3.

Table 3.3. Coverage improvements with 10,000 generated programs

		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%

Result: The coverage improvement information is shown in Table 3.3 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 3.3.

- 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 3.3 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 [98] 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 3.4.

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 4.5. It demonstrates how the line, function, and branch coverages are improved with the increasing number of new tests.

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.

Table 3.4. Augmenting the GCC and LLVM test suites with 10,000 generated programs

		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

- The performance of the coverage improvement pattern for DeepFuzz is similar over GCC-5 and Clang-3.

3.4.4 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 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 }
```

3.5 Limitations

In this study, we have presented an automated fuzzing tool called DeepFuzz, that continuously generates well-formed new C programs for stress-testing production compilers. In this section, we present limitations of our existing work.

3.5.1 Model

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. The structure of statements that used often, e.g. assignments, can be captured in our model precisely and completely but for those features appear very seldom, our trained model might have already "forgotten". We believe that by feeding in more high-quality data, where different patterns or features distribute averagely, the generation pass rate will be improved. 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. For example, previously, we only have "swap(str1, str2);" in the training dataset, but we can create more statements like "swap(str3, str4);" or "swap(str5, str6);". This enlarged training dataset will encode the structure of "swap(*, *);" where * can be replaced by any declared variables, into our model.

On the other hand, because the generation is based on the prefix sequences, it will lose some global information which are out of the scope of the prefix sequence. For example, if we adopt *G3* for the generation and it chops out the ending right

curly bracket for a if statement. If our model does not predict a curly bracket to end, this generated program is ill-formed. 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.

3.5.2 Black-box Algorithm

In our study, the proposed program generation method is based on a *Sequence-to-Sequence* model. Although our prototyping tool has achieved a considerable performance in terms of pass rate, which indicates that our model has encoded the language patterns of C in the training data very well, it cannot be viewed as a representation for the entire C grammar. The trained *Sequence-to-Sequence* model is a black box to us.

By reviewing existing literature, we find some methods to explain black-box algorithms, e.g. *Sequence-to-Sequence* [3]. The most popular method to visualize high-dimensional vectors is to project them into two-dimensional space using t-SNE [66]. Li et al. [40] explored the syntactic behavior of an RNN-based sentiment analyzer, including the compositionality of negation, intensification, and concessive clauses, by plotting a 60-dimensional heat map of hidden unit values. Specifically, we notice that researchers also analyzed whether the black-box model can learn source

syntax after the training process [81]. They explain in their work that after training the *Sequence-to-Sequence* model with natural language sequences, low-level syntax characters like Part-of-Speech (POS) tags are encoded in the model. As a future study, we can follow a similar method but focus on our domain, relying on which, we can explain what syntax patterns of C are learnt by the model and what are lost.

3.5.3 Generation

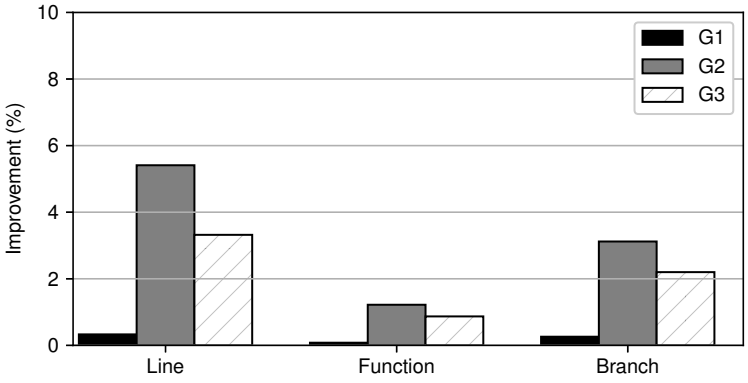
In our study, in order to generate programs to achieve a higher pass rate, we only generate two new lines of code and place them back into a seed program. From the evaluation result, we draw the conclusion that the generation strategy is the most influential factor in this compiler fuzzing task. Although we get a good result in the fuzzing job which makes a remarkable improvement in coverage, it can perform better.

Essentially, by merely modifying the control and data dependency of test programs will increase the testing coverage for compilers quickly [103]. For our next step, we can try to generate programs more progressively, e.g., generate more than two new lines and insert them into more locations. We can conduct a comparative study among different generation strategies and see how the difference can contribute to testing coverage based upon our current setting.

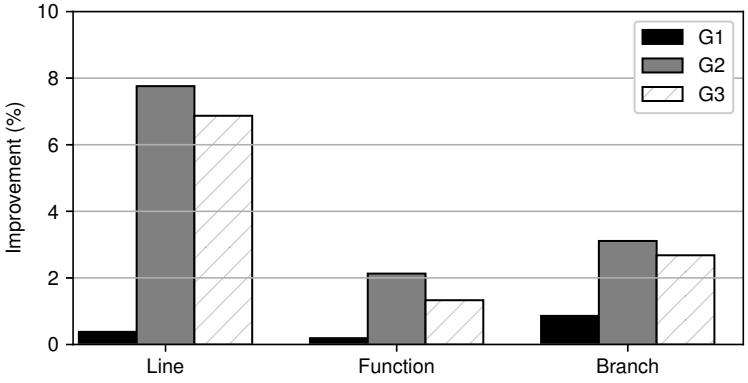
3.6 Summary

Compiler testing is critical for assuring the fundamental correctness of computing systems. Fuzzing is one of the mainstream technologies to assist with compiler testing. In this study, we proposed an automatic grammar-based fuzzing tool called DeepFuzz which learns a generative recurrent neural network that continuously provides syntactically correct C programs to fuzz off-the-shelf compilers, GCC and

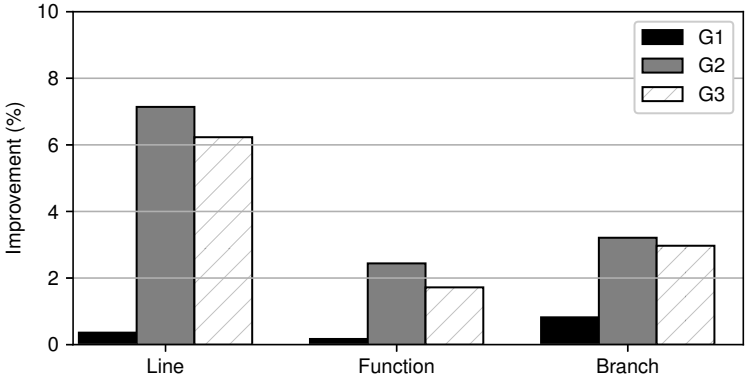
Clang. We conducted a detailed study on analyzing how key factors, i.e. sampling method and generation strategy, affect the accuracy of this generative model and how different improvements of testing efficacy are achieved. DeepFuzz generated 82.63% syntax valid programs and improved the testing efficacy in regards to line, function and branch coverage. With the preliminary evaluation, we found and reported 8 bugs in GCC, all of which are actively addressed by developers.



(a) NoSample

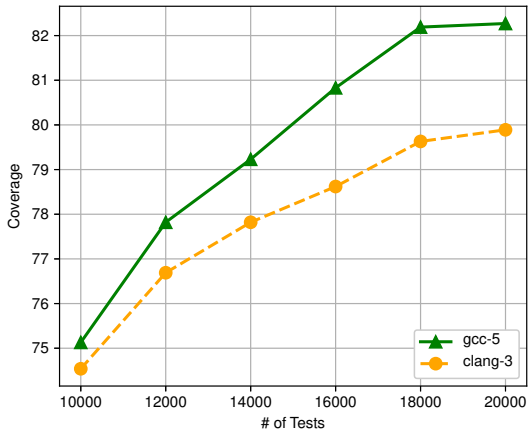


(b) Sample

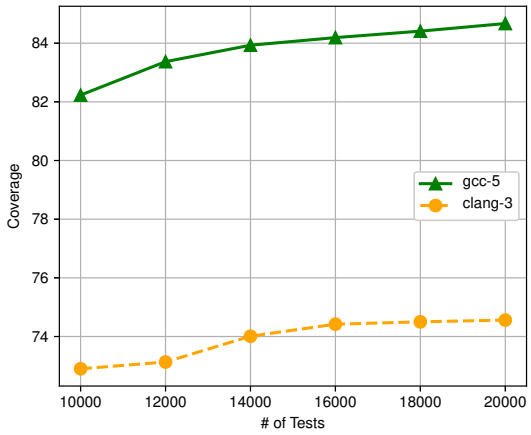


(c) SampleSpace

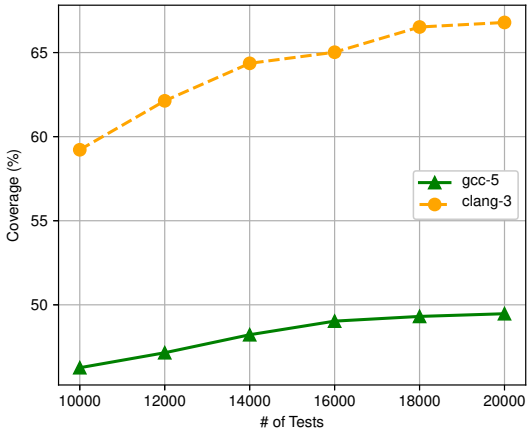
Figure 3.3. Coverage improvements for different sampling methods



(a) Line Coverage



(b) Function Coverage



(c) Branch Coverage

Figure 3.4. Coverage improvement with the new tests generated

Chapter 4
Program Synthesis based on Reinforcement Learning for Compiler Fuzzing

Testing is widely used to assure software quality. However, automatic generation of high-quality test suites is challenging, especially for software that takes in highly-structured inputs, such as compilers. Compiler testing remains difficult, while a substantial amount of research is focused on trying to generate programs that are syntactically and semantically valid. However, they either depend on human-made grammar or a large data set to learn a machine learning model to represent partial language grammar. They both encounter the completeness problem which is a classic puzzle in software testing. In this study, we propose a reinforcement learning-based approach for program synthesis. A naive model was provided at the beginning, and it evolves with the rewards provided by a target compiler that we are going to test. By iterating the learning cycle, the model learns how to write valid programs and how to generate programs that improve the testing efficacy. We integrated the proposed method into a tool called AlphaProg. We analyzed the framework with four different reward functions, and our study revealed the effectiveness of AlphaProg for compiler testing. We performed an in-depth diversity analysis of the generated programs, which explained the improved testing coverage of our target compiler.

We reported two important bugs for this production compiler, and they have been confirmed and well-addressed by the project owner.

4.1 Problem

Compilers are the most import components of computing systems. Although vast research resources have been deployed to verify production compilers, they still contain bugs, and their quality need improvements [84]. Different from application bugs, errors in compilers are usually harder to find, which are not the first place to put breakpoints when a developer tries to debug an unexpected behavior during compilation. They are presumably correct for most application developers, though a simple bug can be exploited for backdoors, which has been demonstrated by researchers [20]. Therefore, it is critical to enforce the validity of compilers with more advanced techniques.

Testing has been widely adopted [12, 79] to verify the correctness and robustness of compilers, and random test case generation, fuzzing, has proven to be an effective method of improving testing efficacies and detecting software bugs [11, 43], including blackbox fuzzing and whitebox fuzzing. The main difference between fuzzing and blackbox testing is that fuzzing focuses on program crashes and hangs, though the testing is more general, which aims at detecting types of syntactical and semantic errors with well-defined sanitizers. Although blackbox fuzzing is efficient for general software, existing techniques are not applicable in for compiler testing that includes highly-structured inputs.

To compile a program, there are a few stages, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation. If a generated test program is not valid, it will be rejected by exception handling in the early stages, such as during the lexical analysis, and the early

rejection will prevent it from testing a deeper site of compilers, which is in contrast to software testing. To generate high-quality inputs in the context of compiler testing, there are two mainstream tracks in existing blackbox fuzzing methods, including mutation-based [68] and generation-based [97]. Mutational approaches start with a few seed inputs and rely on simple mutations, such as bits flip, replace, insert, and delete, depending on designed heuristics [101]. The design rationale focuses on exploring the entire input set by mutating local structures while maintaining global structures from the seed set. Many of the mutated programs are valid in terms of syntax and semantics since only a small part of originally valid programs are changed, and it is more efficient for compiler testing where validity is of greater importance. However, the effective ratio is not good enough, and accordingly, researchers have proposed more rigorous generation engines that encodes formal language grammar for whole program generation [98]. Typically, they conform both syntactic and semantic rules for generating effective programs for compiler testing. However, it takes human effort to construct the grammar-based generation engines, where only a subset of the whole language grammars are encoded as claimed by most of the owners of fuzzing engines in this type. To reduce human labor, researchers have proposed the use of deep neural networks to learn language patterns from existing programs [18, 28, 60]. Based on a sequence-to-sequence model, language patterns can be acquired in terms of production rules and then used for new program generations. The neural networks can capture most syntactical features and generate new tests, which are valid and effective, while no human effort is required to construct the grammar. But their successfulness depends on the chosen data set, which is used for fitting the model and as the seeds. Without such a valid and diverse dataset, which is usually the test suites built by programmers, the proposed machine-learning-based approach does not work as expected.

In this study, we addressed the problem of building a deep learning-based approach without using any datasets for learning. We developed a reinforcement learning framework (*deep Q-learning*) to bootstrap the neural nets that encode language patterns from scratch with the objective of returning messages and runtime information during compilation. Starting from an artificial neural network with random weights, we generated new programs within a limited time period. We then asked any production compiler to compile such a program and collect both the returning message and runtime information, i.e. execution traces, to provide a reward, which is used to train the neural network. Gradually, with more programs generated, the neural network will be trained to generate programs according to our expectations. To achieve better performance regarding compiler testing efficacy, we constructed coverage-guided reward functions to balance the program validity and testing coverage improvement of target compilers. In such a manner, the trained neural network will eventually learn to generate programs that are valid and diverse.

We built the proposed framework into a prototyping tool called AlphaProg. We deployed AlphaProg on an esoteric language called BrainFuck [77] (we use BF in later context), which is a Turing-complete programming language that only contains eight instructions. We explored the effectiveness of AlphaProg by testing an industrial-grade BF compiler called BFC [35]. We compared AlphaProg results under four different reward functions for compiler fuzzing, and AlphaProg performed well in terms of validity and testing efficacies. We also describe the dynamics of generated programs and discuss the evolving process of the trained model from the perspective of program diversity. During the analysis, we also detected two important bugs of this target compiler that they were actively addressed by the project owner in the new released version [56].

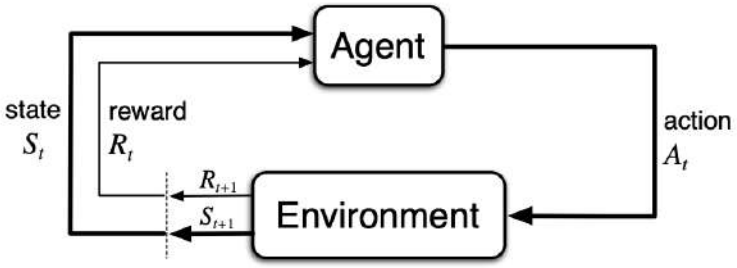


Figure 4.1. The agent-environment interaction in a Markov decision process. Reprinted Reinforcement learning: An introduction (p. 38), by Richard S. Sutton and Andrew G. Barto, 2017, MIT press. Copyright 2014, 2015, 2016, 2017. Reprinted with permission. [87]

4.2 Overview

The Markov Decision Process (MDP) is a discrete time stochastic control process that conforms the Markov property, which states that "the future is independent of the past given the present" [67]. The proposed program generation task can be modeled as an MDP which sees a program as a string of characters in this language, and in each step, a single character is generated. In this section, we elaborate how we modeled the generation of BF code for compiler fuzzing as an MDP and how to fit the constructed reinforcement learning framework.

4.2.1 Program Generation

MDP: A Markov decision process is a 4-tuple (S, A, P_a, R_a), where S is a finite set of states, A is a finite set of actions and it is a transition between two states. And for each different state s, the probability of taking action a is $P_a(s, s')$, accordingly, it receives an immediate reward $R_a(s, s')$, where s is current state and s' is the state after action. Figure 4.1 shows the agent-environment interaction in a Markov decision process. It shows one complete interaction between the agent and interaction. Starting the iteration from the agent, one action $A_t \in A(S_t)$ will be selected and

performed. Once the environment receives the current state S_t and action A_t, it responds with a numerical reward R_{t+1} and finds the agent a new state S_{t+1}. Therefore, the MDP agent will handle the decision making in a sequence that looks like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{4.1}$$

If we see a program as a string of characters of such language, which can normally be any ASCII characters, then program generation is the process of appending new characters from an empty string to an *EOF*. The generation of an *EOF* may vary and a simple implementation is set *EOF* at a fixed point. That is to say, we limit the length of generated programs. According to literature [84], the test cases revealing bugs for C compilers are typically small, with 80% of them being fewer than 45 lines of code. In this case, our goal becomes generating a fixed length of strings compilers used for execution. The core problem of MDP is to find a policy π for making action decisions on a specific state s. That is an update of the probability matrix, $P_a(s, s')$, which achieves the optimal reward $R_a(s, s')$. In the fuzzing task, the probability for each transition will be learning by neural networks to achieve an optimal reward which combines two important metrics (1) the validity of generated programs and (2) compiler testing coverage. The validity of generated strings will be confirmed by returning messages of compilations and it demonstrates how the policy conforms formal language production rules. Traditional production rules are token-based and usually represented in their simplest form by shifting and reducing sentences such that language can be parsed. In our scenario, the policy π is a redundant version of production rules, and for each sentence and each next character, there exists a specific rule to follow with. Moreover, for compiler testing coverage, it will be calculated by analyzing the runtime information of each compilation. We will illustrate details about reward functions in Section 4.3.2.

forward	::=	'>'
backward	::=	'<'
increment	::=	'+'
decrement	::=	'−'
put	::=	'.'
get	::=	','
while	::=	'['
wend	::=	']'

Figure 4.2. The BF language

BF Language: The BF language is an esoteric programming language [77] that contains only eight instructions. Although the language constructs are simple, the language is fully Turing-complete. The eight commands in BF are detailed in Figure 4.2. All characters other than those in the table should be considered comments and ignored. The basic idea behind BF language is memory manipulation that provides an array of memory blocks initiated with a value of zero where you can more the memory pointer forward or backward and increase or decrease memory values. To input or output ASCII characters, it has specific symbols to put and get and for writing more expressive code, it contains *while* and *wend* (which is short for while end) to open and close loops. The BF language is context-free and it has one-to-one mapping language structs to the C programming language which limits the operations on arrays.

To generate code in the BF language, the MDP problem is concretized as appending one of the eight instructions at each step. By limiting the program length to 50 characters, the task becomes extending an empty string until it reaches a length 50 and by querying a production compiler of BF and recording execution traces to calculate coverage information. We can draw conclusions regarding whether the generated strings are of high quality in terms of compiler testing. To compile code in the BF language, people build compilers that usually interpret the BF code into

C language or LLVM IR for optimization. In this study, we targeted BFC [35], an industrial-grade BF compiler. This compiler first parses the source code first into its own IR called BF IR for optimization. It provides a range of peephole optimizations including combining instructions, loop simplification, etc. The optimized BF IR is then compiled to LLVM IR and then handled by the LLVM infrastructure to generate *x86* executables. During the compilation, it will report syntax errors such as unclosing while-statement by showing *no matching* [. It will also report warnings including "pointer out-of-bound" and "no-effect instructions". This is the BF compiler that mostly used on Github and is still actively under maintenance.

4.2.2 Reinforcement Learning

Traditionally, reinforcement learning describes the cycles of interaction between an agent and an operational environment, and gradually, an optimal policy can be learned by trial-and-error for sequential decision-making problems [5, 87]. Since we modeled the program generation task as an MDP, we can apply reinforcement learning frameworks to achieve the policy to generate BF programs character by character. Therefore, we build an off-policy and model-free reinforcement learning process that attempts to figure out the value functions directly from the interactions with the environment. We use Temporal Difference Learning (TD-Learning) for our policy learning task that at each step t, an estimate of reward is given to update the value function where actions are chosen via selecting among the ones have the highest value. One of the most important breakthroughs of reinforcement learning is *Q-learning*, which is an off-policy TD control algrithm [95]. It estimates a state-action value function for any provided policy that selects actions of the highest value. Taking in a concrete pair of examples for the current state and a deterministic action, by querying a value table, which is called Q-table, we can obtain an estimation of the

Algorithm 1 Reinforcement Compiler Fuzzing

Output: action-value function Q-network
initialize Q-network arbitrarily, randomly assign the weights
for for each episode e do **do**
  encode state s from empty string
  repeat
    $a \leftarrow$ action for s derived by Q-network, e.g., ϵ-greedy
    $s' \leftarrow$ append a new character on s
    $r \leftarrow$ calculate reward from runtime traces
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)$
    $s \leftarrow s'$
  until state s is a terminal state
end for

value. By comparing the values for each of these pairs, we can select the action that has the highest value under such a state.

The usage of Q-learning for MDP with large state space is the *Deep Q-Learning* [88]. Different from traditional *Q-learning*, it replaces the Q-table with a deep neural network. Accordingly, this neural network takes the input of a current state and outputs a value for each action in the action space. The predicted action will be selected with a *softmax* function from the calculated value table. The detailed process is shown in Algorithm 2. At the very beginning, we have an arbitrary neural network with randomly assigned weights. When starting an episode, we encode a given state, i.e. an empty string at first, into the Long-Short term memory (LSTM), which is a variant of recursive neural network (RNN). It will encode variable-length strings into a fixed-length vector that contains all the features of such strings. The Q-network will be used for predicting values for each action at one step, that is, based on a given string, a new character will be chosen to be appended to the original string. As long as we have this new character being selected, we have the string after taking action, and we calculate the reward for this state-action from runtime information. This reward is used to update the Q-network, that is, the Q-network is improved

round by round. We end an episode while a terminal state is achieved, and in our preliminary design, we cut the episode by a fixed string length. We will detail the model configuration, reward calculation as well as the training process in Section 4.3.

4.3 Model

We proposed a reinforcement learning framework based on Q-learning to generate the BF code for fuzzing BF compilers and we show the overall generation process in Figure 4.3. In this framework, there are essentially two main components, the fuzzing agent and the environment. The fuzzing agent will try to generate a new program with best practice, i.e. the provided neural network, and the environment, i.e. the compiler, will provide a scala reward for evaluating this synthesized program. To generate a new program, that is a sequence of characters in our context, the neural network will take in a base string x_t for the prediction, character by character. The generated program y_t is a new string by appending a new character to the base string. The quality of this new program will be evaluated and a scala reward r_t calculated with the message and execution trace from the compilation will be provided for training the neural networks which are initialized with random weights and will evolve gradually with more strings are generated and evaluated. In this section, we will detail the model configuration we choose and elaborate on the reward function we defined.

4.3.1 Action-State Value

In traditional Q-learning, there is an action-state value table, which we call Q-table, for querying a value for any predicted actions given a current state. To improve the model to be applicable for tasks with large state-action space, the Q-table is replaced with the deep neural network.

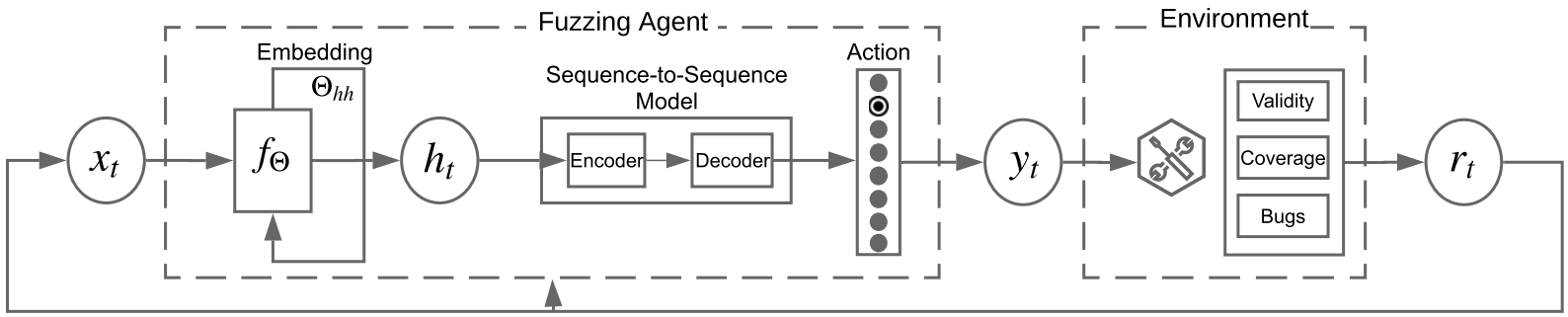


Figure 4.3. Compiler fuzzing process (generative)

In our design, observing a current state, that is a string of characters, the action-state value network should predict an action for the next step. The action is to select a character from the BF code language to append. Valid BF code contains eight different characters as described in Section 4.2.1. We choose a variant of RNN, the Long-Short-Term-Memory (LSTM) [34] for sequence embedding. Recurrent neural networks are designed for sequence predictions for embedding variable-length sequences into feature vectors. LSTM works better for longer sequences, such as text in paragraphs, by remembering the cell state along the connected recurrent neurons in one layer. We use a LSTM layer with 128 neurons for sequence embedding connected with the two LSTM layers with 512 neurons respectively. The output layer is a dense layer with 8 neurons activated by a linear function and it allows the predicted action with the highest value to be output.

4.3.2 Reward

The reward function is key to reinforcement learning frameworks that indicates the learning direction. In the compiler fuzzing task, there are two main goals: (a) the generated programs should be valid; (b) the generated programs should be as diverse as possible. For validity, the generated programs are supposed to be both syntactically and semantically valid. There are a few stages during the compilation process and if the test code is rejected during early stages, such as the syntax analysis, the compilation will be terminated and the rest execution paths won't be tested. Thus, the validity of generated test programs is important for the fuzzing task. In addition to validity, diversity is another goal we want to achieve from the perspective of testing efficacies. If similar tests are generated, although they are valid to be successfully compiled by target compilers, we can not acquire improved testing coverage, thereby hinder the potential for AlphaProg to become a tool for fuzz testing. In other words, we prefer more legal language patterns are explored

and encoded into the neural networks other than periodically synthesizing test code with same patterns.

In our design, we set up four different reward functions for the learning process which demonstrates the two different learning goals and how to achieve the balance in between. First, considering the syntactic and semantic validity, we set the reward function as

$$R_1 = \begin{cases} 0, \text{ length is less than limit} \\ -1, \text{ compilation error} \\ 1, \text{ compilation success} \end{cases} \tag{4.2}$$

That for any intermediate programs during a generation episode, we give it a reward of 0 until its length hits our limitation. To verify the validity of a synthesized program, we use a production compiler to parse this program and then judge its correctness based on the compilation messages.

Compilation Message: Usually, there are five kinds of compilation messages: *no errors or warning*, which means the program is successfully compiled to executables without any conflict to the hard or soft rules defined by the compiler; *errors*, which means the program does not conform syntactic or semantic checks and hits the exceptions that terminate the compilation process; *internal errors*, which means the compiler does not conform pre-defined assertions during the compilation and it indicates an error (bug) of the compiler; *warnings*, which means the compilation succeeds but there are some soft rules have not been met, such as that the program contains some meaningless sequences; and *hangs*, which means the compilation falls into some infinite loops and it does not exit in a reasonable time. We consider three cases among these compilation messages as the indicator for a valid program: *no errors or warning*, *warnings*, and *internal errors*. Theoretically, this reward function should guide the model to synthesize programs that are valid with least

effort, such that, it can be repeatedly generating the same character all the time in the synthesized program.

Second, considering the diversity of the synthesized program, we can use the newly covered basic blocks on the compilation trace as the testing coverage. It shows how many different language patterns have been covered in the generated testsuite. Therefore we have,

$$R_2 = B(\mathrm{T}_p) / \bigcup_{\rho \in I'} B(\mathrm{T}_\rho), \tag{4.3}$$

as the reward. In this new reward function, $B(\mathrm{T}_p)$ is the number of newly tested unique basic blocks of the execution trace for a program p and $I' \subset I$ is all the programs generated so far in this test suite. It makes the reward a continuous scalar value in the range of $[0, 1]$, where 1 is achieved when all the basic block on the compilation trace have not been tested forehead, and 0 is achieved when an existing program is generated repeatedly or all the basic blocks on the compilation trace have been executed so far.

Third, we adopt a combination of reward for validity and diversity of generated programs. To achieve a balance, we have

$$R_3 = \begin{cases} 0, \text{ length is less than limit} \\ -1, \text{ compilation error} \\ 1 + B(\mathrm{T}_p) / \bigcup_{\rho \in I'} B(\mathrm{T}_\rho), \text{ compilation success} \end{cases} \tag{4.4}$$

as the reward function. For all the generated programs that are compiled successfully, we use the portion of newly tested basic blocks as the reward. For the other two cases, we still return reward 0 when the program length does not hit the limit, and -1 when the program is not compilable. This reward function will be partially continuous which is similar to an activation function. When a program is valid in terms of syntax and semantics, the reward should be a value in the range of $[1, 2]$,

where 2 is achieved when all the basic blocks are new, and 1 is achieved when a valid but repeated program is generated. This reward function motivates the program generation towards the training purpose, that is to generate valid, and then diverse programs for fuzzing target compilers.

Fourth, based on the previous reward, we add the control-flow complexity of synthesized programs into consideration. According to Zhang et al.'s study [103], the increase of control-flow complexity of programs in the testsuites will remarkably improve testing efficacies of corresponding compilers. Effective testing coverage can be improved by 40% by simply switch the positions of variables in each program within the GCC testsuite. In our design, we add the Cyclomatic Complexity of synthesized programs into the reward function. It is one of the representations for describing program control-flow complexity.

Cyclomatic Complexity This value is one of the software metrics to measure the quantitative complexity of a target program by counting the number of linearly independent paths [96]. The complexity M is defined as $M = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of connected components. We calculate the complexity of each generated program and display the value for all the valid programs generated from the training process. To calculate the cyclomatic complexity, we first dump the converted LLVM IR from the compilation process. Then we develop a plug-in to generate this number based on the control-flow graphs from the LLVM optimization tool.

That we have,

$$R_4 = R_3 + C(p)/max(C(\rho : \rho \in I')). \tag{4.5}$$

In this function, $C(*)$ is the cyclomatic complexity of a program. We simply add the cyclomatic complexity of a synthesized program divided by the max value we get till now in the previous reward function R_3. In other words, if the synthesized

program does not hit the length limit, we give it a reward of 0 and if it is not valid, we give it reward -1. Otherwise, the reward will be a combination of program validity, testing coverage, and program control-flow complexity. When a program is valid within the compilation process, it will be given a reward in the range of $[1, 3]$, where 3, the max value, is achieved when all the running basic blocks are new and it has the most cyclomatic complexity of synthesized programs till now. This reward function motivates the program generation towards our training purposes as well, which is to generate both valid and diverse programs. In addition, it takes the program complexity into account which indirectly improve the testing coverage.

4.3.3 Training

During the training stage, we bootstrap the deep neural network for program generation that takes in as input a current program $x \in S(x)$, the action a that generated x from a previous state x', the reward $r \in [-1, 2]$ that is calculated based on compilation, and an original Q-network. On a given state, this Q-network predicts the expected rewards for all defined actions simultaneously. We update the Q-network to adapt the predicted value $Q(x_t, a_t)$ according to the target $r + \gamma max_a Q(x_t + 1, a)$ by minimizing the loss of the deviation in between. For all actions in the action space other than the predicted one, they are updated by zero loss. The convergence rate of the Q-network is determined by the hyper-parameters, i.e. the learning rate of stochastic gradient descent during back propagation as well as the choice of γ, which is a discounted rate between 0 and 1. A value closer to 1 indicates a goal that is targeted on long-term reward while a value closer to 0 means the model is more greedy. We adopt the ϵ-greedy method in the training process to balance exploration and exploitation, that with probability ϵ, our model will choose a random action and with probability $1 - \epsilon$ it will follow the prediction from a neural network. In the implementation, we make the value for ϵ decaying, that at earlier stages of

training, the chance to choose a random action is higher but the probability goes down proportionally to the number of predictions. It indicates we gradually rely on the trained neural network other than based on random guesses to explore.

4.4 Experiment

We propose a reinforcement learning framework to generate BF programs for fuzz testing BF compilers. To evaluate our prototyping tool AlphaProg, we perform studies on training the model towards the two different goals by setting reward functions as described in Section **??**. We will log the valid rate and testing coverage improvement during the learning process. The analysis will confirm our guess on the leading role of the two reward functions. To demonstrate the testing ability, we compare our tool with random fuzzing with 30,000 newly generated programs, in terms of testing efficacy. To elaborate its effectiveness on generating more diverse programs, we also study the generated programs to explain the evolving process of the training model. In this section, we will report the detailed implementation of AlphaProg, and will also discuss the experiments we conducted.

4.4.1 Implementation

We build AlphaProg by applying an existing framework of binary instrumentation and neural network training. The core framework of the deep Q-learning module is implemented in Python 3.6. In our implementation, the program execution trace is generated by Pin [65], a widely-used dynamic binary instrumentation tool. We develop a plug-in of Pin to log the executed instructions. Additionally, we develop another coverage analysis tool based on the execution trace to report all the basic block touched so far. It will also report whether and the number of new basic blocks are covered by a certain new program in the compiler code. Additionally,

Table 4.1. Valid rate with different rewards

# of Test (K)	5	10	15	20	25	30
Reward 1 (%)	170	180	210	1000	1000	1000
Reward 2 (%)	16	19	41	182	235	221
Reward 3 (%)	21	48	102	312	479	431
Reward 4 (%)	21	48	102	312	479	431

our environment will also log and report abnormal crashes, memory leaks or failing assertions of compilers with the assistance of internal errors messages.

The exploration, exploitation trade-off is a dilemma that we frequently face in reinforcement learning. On the decision-making process, exploitation means the model will make the best decision on a given current state and exploration aims at gathering more information. In our program generation problem, exploitation is to take advantage of a trained model to generate sequences that conforms program grammar rules as much as possible. And exploration means the fuzzing agent will randomly choose a character to append and it allows the generated sequences to vary. In the implementation, the trade-off between exploitation and exploration is configured by a hyper-parameter ϵ, where ϵ is the possibility that the fuzzing agent takes a randomly selected action rather than taking the action that to maximize reward. And to achieve a balance, we take the epsilon-greedy strategy, that ϵ is a percentage of the time. In this design, the fuzzing agent will prefer exploration at first and decrease the possibility of exploration over time. It mimics the general learning process that at the beginning, the model will try to explore and it will help us to generate more diverse data for the model to learn from. And with the model becoming more matured, our design allows the generation to rely on the trained model. If the language patterns are successfully encoded, the generated programs will be mostly valid, or conform the compiler's grammar.

Our Q-learning module is implemented in Tensorflow [1] using a LSTM layer

for sequence embedding that is connected with a 2-layer encoder-decoder network. The initialized weights are randomly and uniformly distributed within $w_i \in [0, 0.1]$. We choose a discounted rate $\gamma = 1$ to address long-term goal and a learning rate $\alpha = 0.0001$ for the gradient descent optimizer. We assign $\epsilon_{max} = 1$ and $\epsilon_{min} = 0.01$ with a decaying value of $(\epsilon_{max} - \epsilon_{min})/100000$ after each prediction. Therefore, the model stops exploration after episode $20,000$. We will open source our prototyping tool AlphaProg for public dissemination after the paper is accepted.

4.4.2 Validity

Since our first goal is to generate valid programs, we plan to evaluate the valid rate of the generated programs during the training process. We compare our proposed method with four different reward functions. The four different rewards set up two different goals for the program generation: for Reward 1, it only evaluates the validity of programs while for Reward 2, it targets on testing coverage and for Reward 3, it combines validity with testing coverage, lastly for Reward 4, it combines validity with testing coverage and program complexity as well. We report the number of valid program numbers every $1,000$ generated programs in Figure 4.4 and Table 4.1.

Reward 1

Reward 1 demonstrates the learning towards generating only valid programs. From the Figure 4.4 we may find that, with the increasing number of programs generated, the valid rate grows fast and by $20,000$ generated programs, the valid rate reaches 100%. From the generation result, we may find that, once the easiest way to generate a valid program is guessed by a random generation, i.e. „„„„ or »»»», the network converges quickly to this point and stops learning anything. The model trained by this reward function achieves the most ratio of valid programs in the synthesis procedure. It is similar to the human learning process, that repetitive actions are

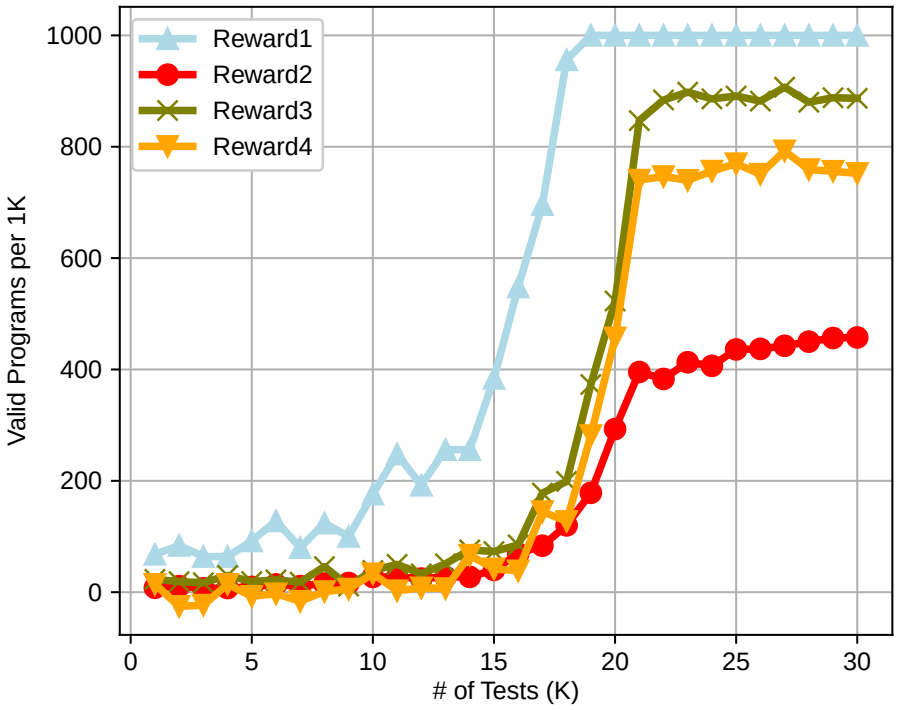


Figure 4.4. Valid rate

easy to remember. We also tried to change the length limit to 80 and 100, the learning process is as fast as when we set the length limit to 50.

Reward 2

Reward 2 demonstrates the learning towards generating diverse programs for improving testing coverage for a target compiler. Without balancing with syntactic and semantic validity, with this reward, we anticipate more diverse programs patterns will be generated but less of them should be valid. From the results in Figure 4.4 we may find that the valid rate stays the lowest for almost all the time which means the generation engine has a low efficiency to learning writing a valid program through the reward on pure coverage. Theoretically, if more are explored on a limited search space, there are chances that the model will learn validity through pure coverage reward since compilation execution traces should be longer for valid programs compared

with those illegal ones. However, due in part to the large search space which leads to less probability to randomly generate valid programs and the positive training samples are far fewer than negative samples. In this case, our model will not be able to efficiently learn any generation strategies out from the training samples.

Reward 3

Reward 3 sets up the goal of combining validity and diversity. In a high-level, to generate valid programs and diverse programs are two opposite goals. To generate valid programs, the model only needs to know one simple way that fits language grammar. For example, in the experiment of using Reward 1, the model only learns that by appending `,` to whatever prefix, it can generate valid programs out of it. However, if the goal becomes generating diverse programs, different characters should be tried which makes validity easy to be broken. The model trained by this reward function achieves the second place in the ratio of valid programs in the synthesis procedure. From Figure 4.4 we may find that the valid rate goes up and down periodically. But from a larger scale, the overall valid rate is increasing and achieves a valid rate approximate to 90% at the final stage. By observing generation results, we find that generating language patterns evolve periodically. For example, the first time a valid program with the local pattern `[>..]` is generated, testing coverage is improved and a relatively high reward is given. But next time when another program is generated with this pattern, testing coverage is not improved any more which leads to a relatively low reward. That encourages the model to try new language patterns while remembering those testified. In addition, the different combinations of learned language patterns also contribute a lot for improving testing coverages as well. We also notice that after trying a new language pattern which fails the validity check, in the next a few episodes, the model will conservatively choose an existing language pattern to generate. We see this process similar as a human cognitive

Table 4.2. Coverage improvement with different rewards

# of Test (K)	5	10	15	20	25	30
Reward 1	39,000	46,000	47000	47000	47000	47000
Reward 2	57,000	59,000	60,000	62,000	69,000	71,000
Reward 3	64,000	78,000	84,000	84,500	87,000	87,500
Reward 4	64,000	78,000	84,000	84,500	87,000	87,500

learning process as well: the learning process is like a spiral which is not improving all the time, especially when the learning tasks are challenging with multiple goals.

Reward 4

Reward 4 sets up the goal of adding program control-flow complexity together with the synthesis validity and diversity. By studying related studies, we know that the control-flow complexity of programs in testsuites is one of the most important factors that improve testing efficacies for compilers. We anticipate the add-on of this factor into the reward function will help us to improve the testing coverage of target compilers while not hindering the program validity that much. From Figure 4.4 we may find that the model trained by this reward function achieves the third place in the ratio of valid programs in the synthesis procedure. The improved pattern is very similar to the pattern under Reward 3 which periodically goes up and down along the way but eventually achieves a valid rate about 80%. The add-on of this value hinders the valid rate a little bit from the model trained by purely relying on coverage. We interpret this as that the goal of synthesizing more complex code does not 100% align with the goal of improving testing coverage. And it is an opposite goal to synthesizing more valid programs as well. Similar to the human learning process, if the learning goals are not 100% aligned, the learning process will be less efficient that may lead to an opposite effect to any one or more of the learning goals.

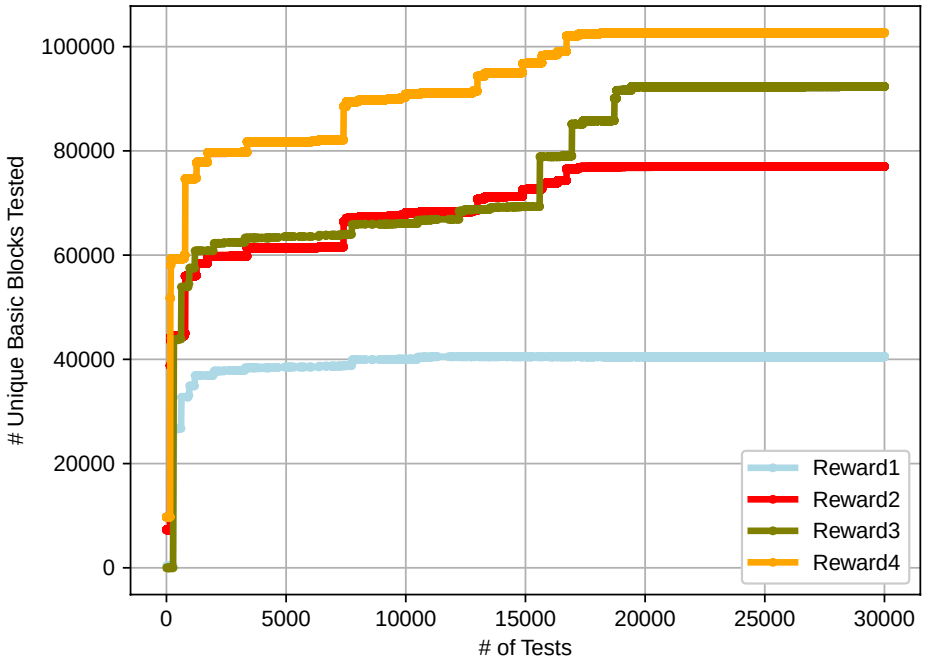


Figure 4.5. Coverage improvement with the new tests generated

4.4.3 Testing Coverage

Coverage improvement is the most important metric for software testing. Traditionally, it denotes the overall lines/branches/paths in target software being visited with certain testcases. In the design of AlphaProg, to improve the performance in this end-to-end learning process, we adopt an approximation to describe the overall testing coverage, that is the accumulated number of unique basic blocks being executed with the generated new programs (new testsuite). A basic block of an execution trace is a straight-line code sequence with no branches except for the entry and exit point in compiler constructions. To capture the overall number of unique basic blocks, we first capture the unique basic blocks $B(\mathrm{T}_p)$ with respect to each execution trace T_p, and then calculate a store of accumulated unique basic blocks number $B(I)$ by union the new basic blocks on current trace with existing ones

that are visited before $B(I')$. In our implementation, we adopt Pintool to log the execution trace of the program compilation, and apply our self-developed plug-in to generate the set of unique basic blocks. We made this plug-in available together with the open-sourced project.

In the experiments, we logged the accumulated testing coverage for the four different rewards we adopted in the framework as well as corresponding scores for each episode. We compare the four coverage improvements and draw conclusions on applying the new reward to balance between the validity and testing efficacy. We show the results in Figure 4.5 and Table 4.2.

Reward 1

The blue line shows the accumulated compiler testing coverage (number of unique blocks tested) by generating programs under Reward 1. With this reward, we may find the coverage improves drastically at earlier stages of training. But it stops growing since episode $11,000$. In the corresponding figure that shows the validity distribution, we also noticed that the valid rate achieves 100% since episode $11,000$ which is very close to the converging point of coverage. It is because our model finally converges at the point that the model keeps producing `,` or `>` for every action. Although the generated programs are 100% valid, they do not improve the testing coverage anymore. This result confirms what we get from the validity test experiment.

Reward 2

The red line shows the accumulated compiler testing coverage by generating programs under Reward 2. With this reward, we may find that coverage improves also drastically but slower than the other two cases at earlier stages of training. It still slowly grows after the improvement stops with Reward 1 but the pace is not as fast as the improvement under Reward 3. The improvement goes on a little slower than the

coverage improvement under Reward 3. In the corresponding figure that shows the valid rate, although, under Reward 2, our model scarcely generates valid programs, these generated one is inspired to be diverse to hit different parts inside the target compiler which eventually improves the testing coverage with lower efficiency.

Reward 3

The green line shows the accumulated compiler testing coverage by generating programs under Reward 3. With this reward, we may find that coverage improves the most drastically at earlier stages and it keeps going high later on until the second-highest testing coverage is achieved eventually. We may also notice that the coverage improves periodically. In the corresponding figure that shows the valid rate, we also observe the regularity of increasing and decreasing wave. In other words, the model is trying to generate valid programs, because the least reward it can get is 1. But it periodically tries to generating some new patterns, usually a combination of existing patterns. In this case, the generated programs can be valid and diverse at the same time.

Reward 4

The orange line shows the accumulated compiler testing coverage by generating programs under Reward 4. With this reward, we may find that the coverage improves as drastically as the synthesis under model trained using Reward 3 at earlier learning stages. The coverage keeps increasing until the highest value is achieved among the 4 designed reward functions. Although the final program valid rate under Reward 4 is lower than those under Reward 1 and Reward 3, but the testing coverage beats those two coverages. To compare testing coverages under Reward 1 and Reward 4, the reason for the better latter case is obvious: the model is trained to achieve better coverage under Reward 4 but not Reward 1. However, to compare testing coverages under Reward 3 and Reward 4, the reason for explaining the higher value under

Reward 4 is more complex. We may interpret it as the side effect of the learning goal of program control-flow complexity. On the one hand, the higher control-flow complexity is a more direct and instant reward to improve the testing coverage. It will trigger the fuzzing agent to generate programs that require more optimizations in the compilation process. On the other hand, it sets up the goal of synthesizing complex program in every episode which is not the goal under Reward 3. Under Reward 3, to improve testing coverage, the fuzzing agent needs to learn new language patterns but under Reward 4, the fuzzing agent needs to additionally learn how to combine the newly learned language patterns in an efficient way because the entire sequence length is limited.

4.4.4 Diversity

We also report the growing traces of control-flow dependencies of the generated programs. In existing studies, researchers developed tools that improved compiler testing efficacies by generating programs with more complex control-flow dependencies [103]. This research indicates the proportional relationship between testcases' control-flow complexity and compilers' testing coverage with limited language patterns. In our experiments described in Section 4.4.3, the results reveal improvements in testing efficacies based on program generation under the three different rewards. In this section, we will explain these improvements from the perspective of generated programs, especially the control-flow complexity of these programs. It also shows a dynamic balance is achieved while balancing the program validity and diversity.

Reward 1

The cyclomatic complexities for programs generated by AlphaProg under Reward 1 scale from 2 to 52, that has an overall median value at 2, which is the lowest among the four different rewards. The value goes up and down until finally stops at a value at 4. It is because the model was trained to synthesize a single same BF program

Table 4.3. Cyclomatic complexity with different rewards

	Min	Max	Median
Reward 1	2	52	4
Reward 2	2	47	11
Reward 3	2	52	11
Reward 4	2	52	18

that is using the character **,** all the time. We may conclude that the learning goal is achieved, which is to synthesize valid programs, in the easiest way. The generation engine starts with random guesses and after some positive samples are generated, it tries to reduce the effort for repeating partial language patterns until finally, it finds a way to always repeat a same character.

Reward 2

Under Reward 2, the cyclomatic complexities for all the generated programs scale from 2 to 47. The overall median is 11 which is higher than that under Reward 1 and the same as that under Reward 3. Because we can only calculate the cyclomatic complexities for valid programs, that are compiled successfully by BFC, therefore, we get fewer data points because we have fewer valid programs. In general, we may find a gradually improved trend of program cyclomatic complexities. We may conclude that the learning goal is also achieved, which is to synthesize diverse programs. Here, we only measure one ax in program diversity. The generation engine also starts with random guesses but different from the case of Reward 1, the model was trained to generate a different combination of existing patterns. Theoretically speaking, we may extend the learning cycles until more language patterns are learned and eventually get an artificial network closer to represent all the rules inside the target compiler.

Reward 3

Under Reward 3, the cyclomatic complexities for all the generated programs scale

from 2 to 52. The overall median is 11 which is the same as that under Reward 2 and higher than that of Reward 1. In this case, we get a few more data points compared with the experiment under Reward 2 because more valid programs are generated. The cyclomatic complexities also gradually trending high across the learning process and which has a steeper incline. We may conclude that the learning goal is achieved, which is to synthesize both valid and diverse programs. The reason for the steeper incline of program complexity may due in part to the higher learning efficiency. Because more valid programs are generated and the number of positive training samples is much more than that under Reward 2, the model has more data to learn from. Therefore, we claim that we may utilize the framework under Reward 3 to continuously generate new programs for compiler fuzzing.

Reward 4

Under Reward 4, the cyclomatic complexities for all the generated programs scale from 2 to 52. The overall median is 18 which is the highest value among the cyclomatic complexities of programs under the four different reward functions. Since this value is only measured for programs that are valid, it is skewed with the valid rate. From the comparison of cyclomatic complexities between Reward 2 and Reward 3, we may summarize that the higher valid rate will lead to more positive learning samples, thereby improves the learning efficiency. In other words, the learning goal will be more effectively achieved. But the valid rate is lower under Reward 4 than under Reward 3. Although, the positive learning samples are fewer under Reward 4, we still get higher cyclomatic complexities. It is because we set the program complexity in the reward function. From another perspective, we may conclude, the most effective way to improve a measurement in reinforcement learning is to directly add this value into the reward function. We may also conclude that the learning goal is achieved the best under Reward 4.

Table 4.4. Synthesis examples

Episode	CC [1]	Program
101	2	`[+, <>++[>..],-+<+[,]-,[<].<-[],>,[>. <[+]]+>«]`
1786	11	`[>[„[... - [<]>+, .+-,. .-.,],].]> .,+[>]>. +..+.`
5096	32	`<-+[. <,[.,-] +]> -.+++<++-.>-,[>.,+,] -<- -[]`
10342	39	`-<[>.<.<.><,]<-<[<.-. ] -,[>- <>++-[],. ]»-+[,<]`

4.4.4.1 Synthesis Examples

To demonstrate how the control-flow complexity of synthesized programs grows, we show four cases that generated during different episodes using the model under Reward 4. The synthesized programs are displayed in Table 4.4 and their corresponding control-flow graphs are shown in Figure 4.6. We draw the abstracted control-flow graphs for them based on the control-flow graphs generated from the LLVM machine-independent optimizer. The original control-flow graphs from the LLVM tool are shown in Figure 4.7, Figure 4.8, Figure 4.9, and Figure 4.10. LLVM represents programs as sequences of instructions in bytecodes and each block in Figure 4.6 is a basic block. For each control-flow graph, we mark the `init` block in dark. The cyclomatic complexities for the example programs are also shown in Table 4.4. We may find that, with the learning goes on, the fuzzing agent learned to synthesize more complex programs which have higher cyclomatic complexities. We display the original control-flow graphs from the LLVM tool for the program examples.

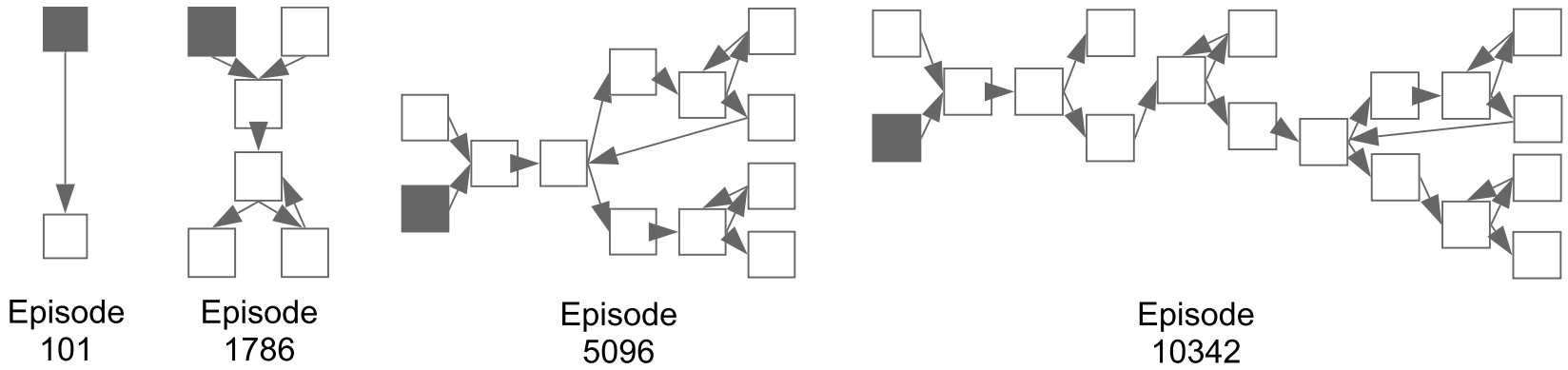


Figure 4.6. Control-flow graphs of synthesized programs (abstract)

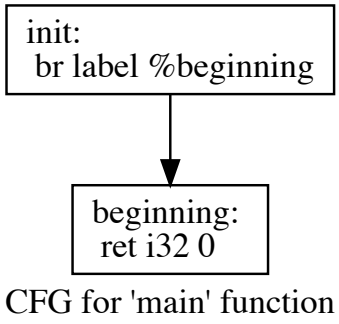


Figure 4.7. Control-flow graph of synthesized programs (1)

Figure 4.8. Control-flow graph of synthesized programs (2)

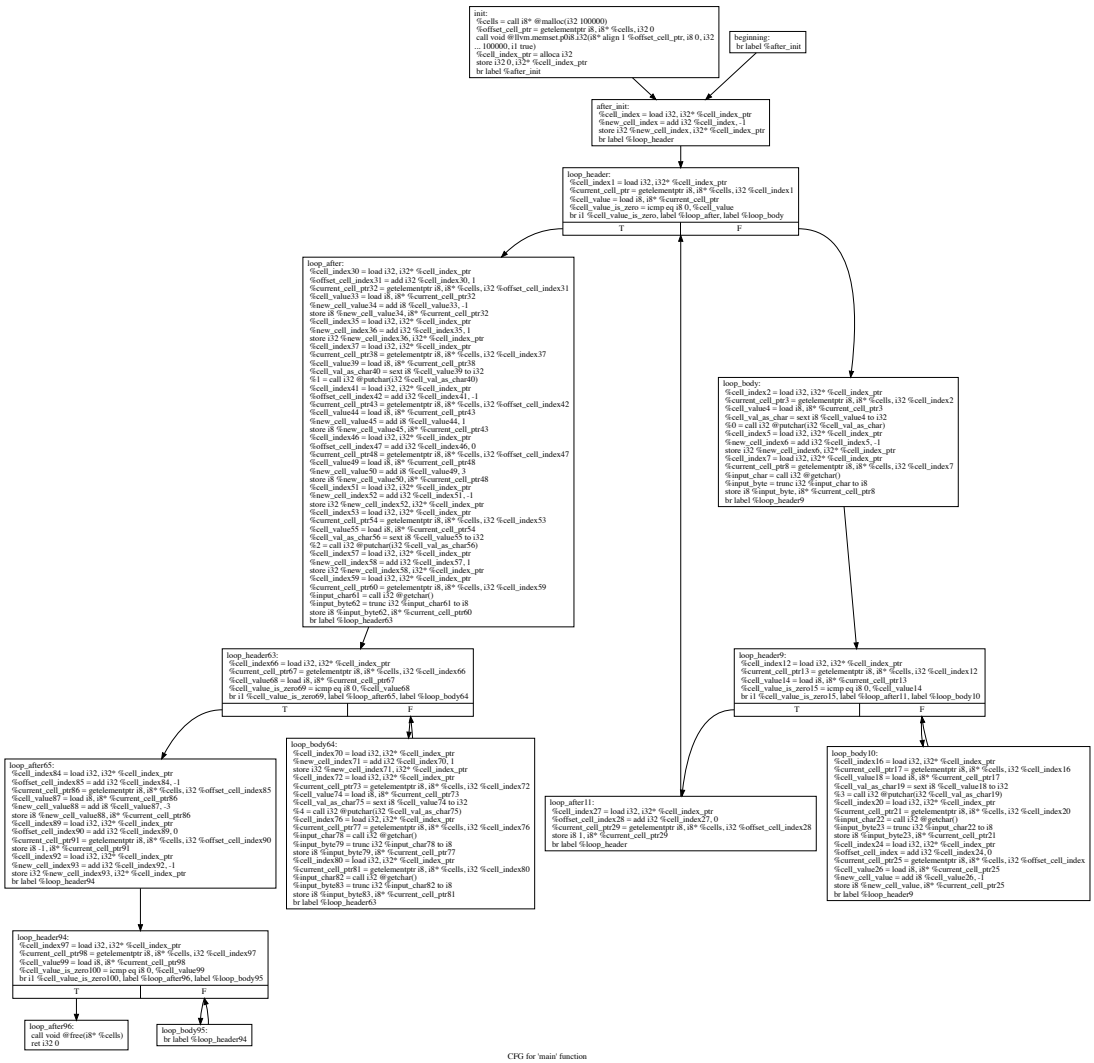


Figure 4.9. Control-flow graph of synthesized programs (3)

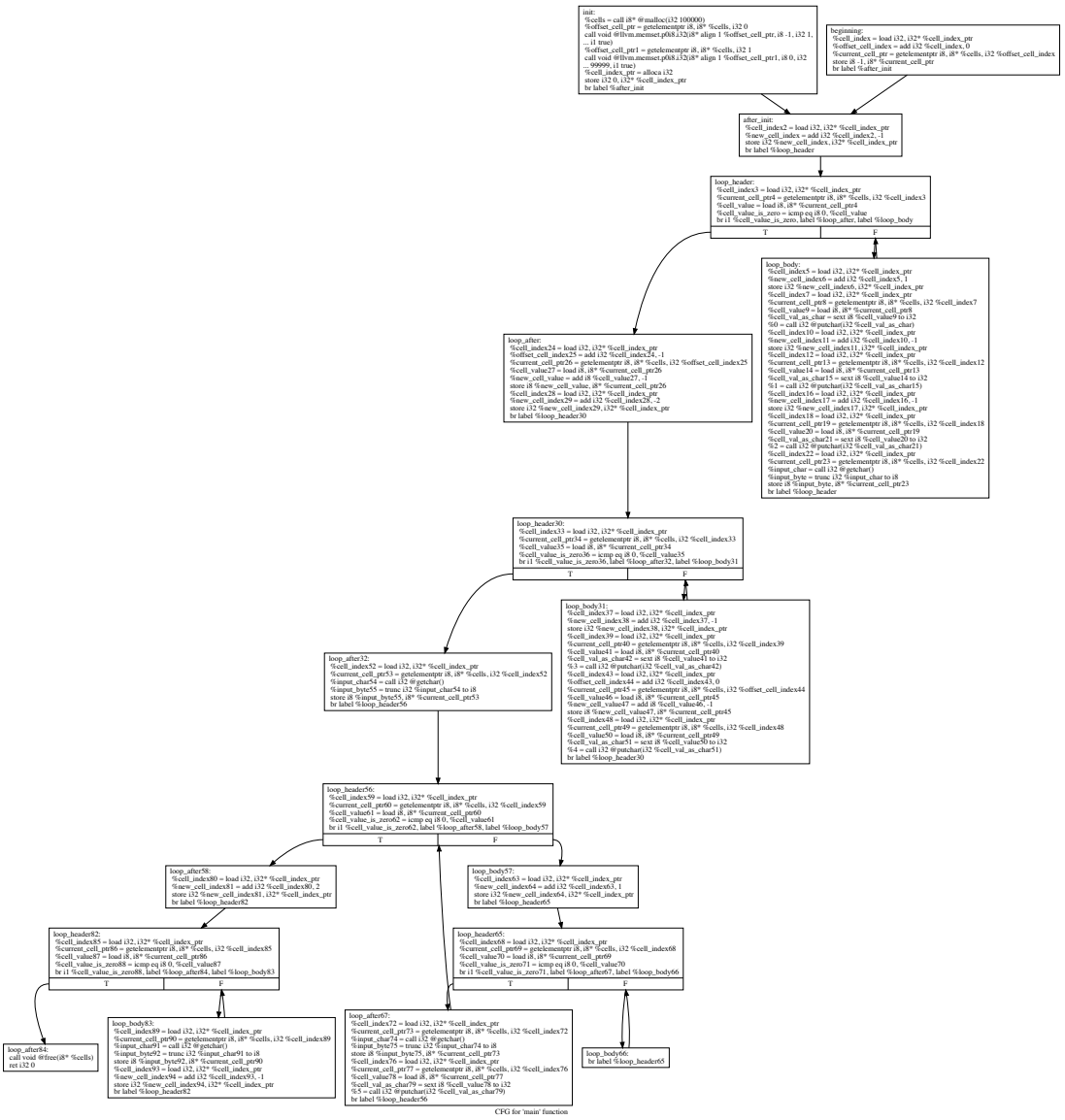


Figure 4.10. Control-flow graph of synthesized programs (4)

4.4.5 Compare with AFL

AFL [101] is a matured production fuzzer that has been widely used for different applications. We also compared AlphaProg with AFL in the two perspectives we focus on: *Validity* and *Coverage*. We use AFL with a single empty seed to generate $30,000$ programs for fuzzing BFC and record the highest valid rate per $1,000$ samples and the accumulated coverage achieved. As a result, the highest valid rate for AFL is 35% and the accumulated coverage in terms of basic blocks tested is $43,135$. It covered 162 paths but found *no* crashes or hangs (actually we ran AFL for 24 hours and *no* crashes or hangs was found). But AlphaProg can achieve the valid rate around 80% under Reward 4 which is the most efficient one for fuzzing, that over 100,000 basic blocks are tested with $30,000$ test samples, and two bugs were detected. With this result, we may claim that AlphaProg is better than AFL in generating valid and diverse programs for compiler fuzzing.

Additionally, we measured the performance of both AlphaProg and AFL by running each tool for synthesizing the $30,000$ samples. We ran the experiment on a server machine with 2.90GHz Intel Xeon(R) E5-2690 CPU and 128GB memory. For AlphaProg, we stopped the training process and timed the synthesis with the trained model. It took 27 minutes for AFL to synthesizing the $30,000$ samples while it took 2 hours for AlphaProg to do the same task. There is still room to improve for AlphaProg in terms of scalability.

4.4.6 Bugs

With the improved testing efficacy, our tool has the potential to discover more compiler bugs compared with pure random fuzzing. And during our analysis, our tool helped is to report two important bugs for the target compiler BFC which is an industrial-grade BF compiler with the most stars (207) and folks (8) on GitHub. We

reported two programs that trigger BFC to hang due to compile-time evaluations [56]. The two programs are listed below,

```
1  . +
2  [ [ [ [ >.
3     [+
4      [<>
5      ]
6     ]
7    ] >
8   ]<>+ .
9  ]>< ,-.,,+++
10 [
11 ]---
12 ]
```

Listing 4.1. Bug 1

```
1  . + . +
2  [ [ [ [ >.
3     [+
4      [<>
5      ]
6     ]
7    ] >
8   ]<>+ .
9  ]>< ,-.,,+++
10 [
11 ]---
12 ]
```

Listing 4.2. Bug 2

The first program triggers the bug during the BF IR optimization and the the second one triggers the bug because the compiler aggressively unroll the loop due to compile-time evaluation, send a huge amount of IR to LLVM, and then it spends ages trying to optimize the IR. Both issues are addressed by the owner of BFC and they confirmed its importance to be fixed later on.

4.5 Limitation

Our work demonstrates the first step using reinforcement learning, *Q-learning* framework for generating valid programs for compiler fuzzing. However, there are still two main limitations in our current work.

First, the existing framework is still not applicable for generating programs in a more complex programming language, e.g. *C*, from scratch due to the large search space. We tried to use our framework to generate C programs token by token, in which experiment we adopted a language space of 141 different tokens and limited the program length to 20 tokens. However, it took days for our prototype to find one single valid C program. The C language structs are difficult to synthesize where the entire search space is 141^{20}, almost $8e + 24$ times of the BF language with same

length limit. We still need more grammars to be encoded in the generation engine to make our framework applicable for languages like *C*.

Second, it is hard to determine the end of a training cycle. Since generating more diverse programs is contradict to generating valid programs, the goal for our reinforcement learning framework is hard to define compared with the game of Go. Currently, the reward functions we defined can inspire the two different goals, yet it is impossible to decide whether the two goals are fully achieved for a training cycle. We still need to improve the design with one generalized goal defined, such as based on the percentage of code covered. However, it is ironic to have this metric calculated which brings a lot of overhead and we need a more in-depth study on this in our future study.

4.6 Summary

In this study, we proposed a reinforcement learning-based approach to continuously generate BF programs for BF compiler fuzzing. With no training data set required, the model was initialized with random weights at the very beginning and it evolves with environment rewards provided by the target compiler we are going to test. With the learning iterations going on, the neural network model gradually learns how to write valid and diverse programs to improve testing efficacies under the three different reward functions we defined. We implemented the proposed method into a prototyping tool called AlphaProg. We detailed the configuration of our model and open-sourced the code. Our study revealed the overall effectiveness of AlphaProg for compiler testing. We also compared metrics under the different reward functions and explained the improved testing coverage by analyzing the generated programs. Our tool helped us to find two important bugs of a production BF compiler, BFC, and all of then are confirmed and well-addressed by the project owner.

Chapter 5
Program Mutation based on Reinforcement Learning for Compiler Fuzzing

Enforcing correctness of compilers is important. Fuzzing is an efficient way to find security vulnerabilities by repeatedly testing programs with randomly modified input data. However, in the context of compilers, fuzzing is challenging because the inputs are pieces of codes that must be both syntactically and semantically valid to pass front-end checks. Moreover, the fuzzed inputs are expected to be distinct enough to trigger abnormal crashes, memory leaks, or failing assertions that have not been detected before. In this study, we proposed an automatic code synthesis framework called FuzzBoost based on deep reinforcement learning. By adopting testing coverage information collected from runtime information as the reward, we propose a learning system with the deep Q-learning algorithm that optimizes this reward. In this way, the fuzzing agent learns the actions to fuzz a seed program that achieves an overall goal of testing coverage improvement. We validated the effectiveness of our proposed approach, and the preliminary evidence shows that reinforcement fuzzing can outperform baseline random fuzzing on production compilers. It also shows that a pre-trained model can boost the fuzzing process for seed programs with similar patterns.

5.1 Problem

Fuzzing is an effective way to find security vulnerabilities in compilers by repeatedly testing the codes with randomly modified inputs. Many existing vulnerabilities are reported by fuzzing techniques [78]. Due to the unlimited search space and limited computing resource, existing fuzzing tools explore efficient strategies in fuzzing program inputs. Especially in the scenario of compiler testing, no one can exhaustively examine the entire input space in practice, or traversing all the possible execution paths of target compilers. Therefore, they typically use fuzzing heuristics to prioritize the fuzzing strategies to be taken. Such heuristics may be random selections or trying to maximize a specific goal, such as code coverage [41], execution timeouts, and crashes [100].

Coverage-guided testing is widely adopted by fuzzers [24, 92, 101], which utilize code coverage as the heuristic for searching a good next fuzz action from a predefined list. These exhaustive bounded searches use domain-specific heuristics and are thus limited in applicability and scalability. Additionally, they do not benefit from *past experiences*, where common knowledge in boosting the fuzzing process across different seeds are shared when similar patterns in the seed files exist. Moreover, most coverage-guided frameworks calculate the rewards/fitness after a single mutation being taken, but which overlooks the power of mutation combinations. State-of-the-art methods, such as American Fuzzing Lop (AFL) [101], incrementally add newly fuzzed programs into the seed set according to defined heuristics after each mutation. However, for coverage-guided fuzzing, testing coverage does not increase linearly. In other words, each of these mutations may not incrementally improve the testing efficacy. They can be rejected by lexical or semantic checks in the early stage of compilation. However, a trace of mutations may trigger a giant improvement because

it may help to generate a valid and different program to cover more paths inside compilers.

The exercising of reinforcement learning inspired the design of FuzzBoost. Reinforcement learning describes the learning process by an agent interacting with the environment to learn an optimal policy by trial and error. It is usually effective for sequential decision-making problems in both natural and social sciences [5, 87]. Theoretically speaking, the problem of compiler fuzzing can be seen as a problem of program synthesis, the goal of which is to cover more paths and trigger more crashes or memory leaks in compilers' execution traces, while compiling such new codes. In this study, we model the compiler fuzzing as a multi-step decision-making process and formalize it into a reinforcement learning problem. Compiler fuzzing is a learning task with a feedback loop. Initially, the fuzzing agent generates new inputs with little knowledge but random heuristics. Then, we let the compiler run with each new input. As the feedback from the environment, we capture runtime information gathered from binary instrumentation technique to evaluate the seed quality according to heuristics we defined in our learning circle. By taking this quality feedback into account, we may construct an end-to-end learning cycle that the fuzzing agent can learn from. By iterating the learning cycle, the agent is trained to generate a new input program to fuzz compilers effectively and efficiently.

Theoretically speaking, the problem of compiler fuzzing can be seen as a problem of program synthesis, the goal of which is to cover more paths and trigger more crashes or memory leaks in the compiler's execution trace while compiling such new codes. In this study, we model the compiler fuzzing as a multi-step decision-making process and formalize it into a reinforcement learning problem. We may see the problem of compiler fuzzing as a learning task with a feedback loop. Initially, the fuzzing agent generates new inputs with little knowledge but random heuristics. We

will let the compiler run with each new input and as the environment's feedback, and for each program execution trace, we capture runtime information gathered from binary instrumentation for evaluating the quality w.r.t. the heuristic we defined for the current input program. For instance, the quality of the generated input can be measured as the number of unique basic blocks on this trace. By taking this quality feedback into account, we construct an end-to-end learning cycle that the fuzzing agent can use to learn. By iterating the learning cycle, the agent can be trained to generate a new input program to fuzz compilers in the most effective and efficient way.

We evaluated FuzzBoost with seed programs from test suites of production compilers, i.e. GCC. To demonstrate the effectiveness of our framework, we also compare it to a baseline system, which applies mutation actions with a uniformly distributed strategy, and it is adopted in widely-used fuzzing tools. FuzzBoost outperforms baseline random fuzzing with a higher coverage improvement on a single seed program. Additionally, to show the generalization of FuzzBoost on boosting the fuzzing process, we design the experiments with seed programs by α-conversion. As a result, our tool has better performance of scalability with a pre-trained model. That means the fuzzing process will be boosted when we reuse an existing model for new seed programs compared with an untrained model.

In summary, we make the following contributions:

- We formalized compiler fuzzing as a reinforcement learning problem by modeling it as a multi-step decision-making process.
- We propose to use deep Q-learning that learns to choose a trace of high-reward mutation actions for any given seed program input. The method is stand-alone and does not rely on any other fuzzing techniques.

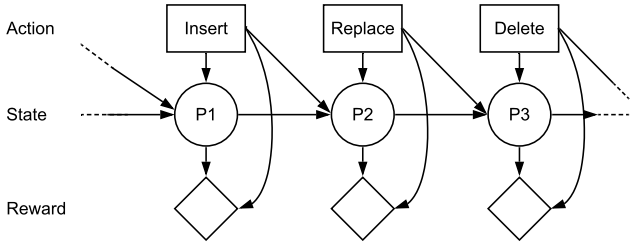


Figure 5.1. Compiler fuzzing process (mutational)

- We implement a prototyping tool called FuzzBoost and analyze real-world fuzzing jobs. It outperforms baseline random fuzzing in terms of testing efficacy.

5.2 Design

Mutation-based fuzzing relies on generating new program inputs by mutating with heuristics based on seed programs. Traditionally, mutation-based fuzzing adopted iterations of one-step fuzzing. In other words, to decide the interest of adding a new mutated input into the seed set, they collect the performance of such input after a single mutation by capturing new crashes in the context of black-box fuzzing or capturing new path information in the context of grey- or white-box fuzzing. However, it overlooks potential performances of a trace of mutations, some intermediate states of which may not be good enough to attract interest or even break the compilation process due to lexical checks in early stages. Therefore, we re-model the problem as multi-step decision-making problem that will give enough attention to these intermediate states being ignored in previous design models. And we formally define the compiler fuzzing and learning process as a Markov decision-making process as described in Figure 5.1.

MDP: As shown in the figure, in this multi-step decision-making process, there is an input mutation engine M, that will perform a fuzzing action a, and subsequently observe a new state x directly derived from the mutated program P_2 by exercising

the predicted action a on an original seed program P_1. This input mutation engine will predict the program rewrites with regard to an extracted state from the seed program. With the given formalization, it is natural to use Markov decision process (MDP) to model this problem. Therefore, the corresponding T-step finite horizon MDP is defined as $M = (s_1, a_1, r_1, s_2, a_2, ..., s_T)$. Here s_t, a_t, r_t represent the state, action, and reward at time step $t = 1, ..., T-1$, respectively. To achieve the trace of most effective rewrites of a seed program, our formalization allows us to apply state-of-the-art reinforcement learning methods, in particular, the Q-learning [95]. We choose to use a variation of Q-learning called deep Q-learning [69, 70] where the value function is replaced by a deep neural network. By training an end-to-end model with stochastic gradient descent to update the weights, we can acquire a well-trained model to perform this program synthesis task and achieve an overall goal to maximize the reward we define.

Q-learning refines the policy greedily with respect to action values by the max operator. Our framework utilizes the deep Q-learning which adopts the deep neural network for the Q function. The algorithm for deep Q learning is presented in Algorithm 2. The Q-network is initialized arbitrarily with random weights at the beginning. During each episode, we use an incrementally trained Q-network for predicting actions in program mutations and retrain the model when we get new rewards for each program state after performing the predicted action. We provide more detailed learning process description in Section 5.3.

Overview: In reinforcement learning, one episode is one complete sequence of states, actions, and rewards, which starts with an initial configuration and ends with a terminal state. In the problem of compiler fuzzing, one episode can be defined as generating a good program by mutating an existing seed program (initial state) with respect to the defined quality and in our preliminary implementation, we hard-coded

Algorithm 2 Reinforcement Compiler Fuzzing

Output: action value function Q-network
initialize Q-network arbitrarily, randomly assign the weights
for for each episode e do **do**
  extract state s from seed program
  repeat
    a $\leftarrow$ action for s derived by Q-network, e.g., ϵ-greedy
    take action a, s'
    calculate r from runtime trace
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)$
    $s \leftarrow s'$
  until state s is a terminal state
end for

the entire trace length of program mutations as one of the terminal conditions (terminal state) as well as define a terminal action that allows the model to end one episode actively. Compare with conventional mutation-based fuzz testing methods, we adopt the same methodology to select a generated input from the seed set where inputs will be continuously fuzzed. The main difference is that, in our design, we *lazy-evaluate* the quality of the fuzzed inputs. Thus, intermediate states that are not valid but eventually turn out to contribute high-quality fuzzed inputs.

Before we start the learning process, we first initialize a randomly generated neural network. In the first episode, State 0 is initiated by preprocessing a seed program P. We initially extract a substring within this seed program with the window size w and offset s. By observing this substring, the neural network will help us to predict a mutation action to be taken. Feasible mutation actions on token-level include *insert* a token, *switch* two or more tokens, *replace* a token, or *change* the window size or offset to enable another substring to observe and mutate. Once an action is decided, we run the compiler (any production compiler) with the program after mutation and calculate the reward r of this new program with a record of the execution trace. The state will move to State 1 after one action being taken. With

the increased number of actions being taken, we deduct the reward by a discounted rate γ which is a value between 0 and 1. We iterate the mutation prediction and evaluation until a *terminal* state. There are four key elements in this process: *state*, *action*, *environment*, and *reward*. We will elaborate on these key elements in our design one by one.

5.2.1 State

A state S is a concrete configuration after each action is being taken. It has similar definition as in the MDP that each process has one state and when the process proceeds, the state updates. Therefore, the state can be the current configuration returned by the environment or any future configurations on this trace. In the case of compiler fuzzing, the agent learns to interact with a given seed program. Therefore, the state is a function about a given input seed program p. In our design, the interaction is performed upon the observation of substrings of consecutive symbols within such an input. Formally, let Σ denotes a finite set of symbols. The set of possible program inputs I in this language is defined by the Kleen closure $I := \Sigma*$. For an input program string $p = (p_1, p_2, ..., p_n) \in I$, let

$$S(p) := (p_{1+i}, p_{2+i}, ..., p_{m+i}) | i \geq 0, m + i \leq n \tag{5.1}$$

denote the set of all substrings of p. We define the states of the Markov decision process to be I and I is a union set of $S(p)$. Thus, we have $p \in I$ denotes an input program and $p_0 \in S(p) \subset I$ is a substring of this input seed program. The entire state space of a seed program is $S(p)$, which is theoretically infinite since any symbol in this language I can increase after mutation. In other words, the seed program can be converted to any other valid programs.

5.2.2 Action

Action A is the set of all possible mutation actions the agent can perform. In most cases, actions are deterministic and should be chosen among a list of possible actions. In compiler fuzzing, we define the set of possible action A of our Markov decision process map extracted substrings $S(p)_0$ to probabilistic rewrite rules. The rewrite rules are defined in accord with the extracted substring and predicted type. In a high-level, we define two types of rewrites, on the extracted content and the extraction window. To be more specific, the rewrites of extracted content are performed on token-level which include *insertion*, *replacement*, *re-ordering*, *deletion* and *replication*. These pre-defined token-level rewrite rules conform with C language lexical requirements. For *insertion*, in we append tokens after the predicted index according to production rules; that is if the last token is an operator, we random sample from existing identifiers for next, etc. For *deletion*, we delete the token located at the predicted index. For *replacement*, we replace the token at predicted index with another token randomly sampled from sets of tokens with same PoS, e.g. if this token is a keyword of C, we select another keyword for replacement. The keyword, operator token set are predefined and identifier token set is generated by parsing the seed file.

The neural network will predict which type and on which position an action should be performed and we employ a lexical analysis on such extracted substring to conduct such mutations on a fine-grained granularity. This will change the input program p_i into p_{i+1} by mutating the substring $S(p)$ in observation, meanwhile, keep the original syntactic and semantic validity with the best effort. For the second type, they are designed to make a change of extraction windows. Atomic mutations include window *left shift* and *right shift*; and window size *up* and *down*, one character length for each. Each of these actions does not modify the original seed program but

motivates an originally extracted substring $S(p_i)$ into another substring $S(p_{i+1})$. For both types of mutations, the time step increases to next state until the termination state incurred on the current episode. The substring rewrites consider every substring in the seed program and predict accordingly to maximize the accumulated rewards along mutation traces. We also define a *terminate* action to early stop the mutation episode. That is to say, the mutation agent can actively terminate a mutation episode while observing an extracted substring.

5.2.3 Environment

The environment is the world that the agent evaluates each action. The environment will take the current state and predicted action as the input and then outputs the reward and next state after executing the action. In compiler fuzzing, the environment is the compiler or verifier. To observe more detailed information about the fuzzing efficacy, we develop a tool based on program execution traces. In this respect, we record dynamic traces when running any production compilers, i.e. GCC, with generated programs. In compiler construction, a basic block of an execution trace is defined as a straight-line code sequence with no branches except for the entry and exit point. We capture all the unique basic blocks $B(\mathrm{T}_p)$ with respect to each execution trace T_p, and calculate a store with all the unique basic blocks covered by the existing test suite I'. In our implementation FuzzBoost, the program execution trace is generated by Pin [65], a widely-used dynamic binary instrumentation tool. Pin provides infrastructures to intercept and instrument the execution trace of a binary. During execution, Pin will insert the instrumentation code into the original code and recompiles the output with a Just-In-Time (JIT) compiler. We develop a plug-in of Pin to log the executed instructions. Additionally, we develop another coverage analysis tool based on the execution trace to report all the basic block touched so far. It also reports whether new basic blocks are covered by certain mutated program

and the number of new covered blocks as well. Additionally, our environment will also log and report abnormal crashes, memory leaks or failing assertions of compilers with the assistance of internal errors alarms from the compiling messages.

5.2.4 Reward

Rewards provide evaluative feedbacks for an RL agent to make decisions. However, rewards may be very sparse so that it is challenging for the agent to learn any algorithms. In the game of Go, a reward only occurs at the end of a game. In such cases, the learning process will converge slowly because of the sparse motivations. Therefore, to design a good reward function to facilitate learning and maintaining the optimal policy is very important. Generally, in fuzz testing, proposed heuristics are program coverages, new crashes, timeout, etc. It aims at enlarging the analyzed surface in target programs being fuzzed and digging into program traces accordingly that are more spurious. In compiler fuzzing, we adopt testing coverage as the reward to motivate the learning towards a search for more areas in the compiler's code. However, not as conventional definitions for coverage, which are usually line/function/branch coverages that require more computing resource to calculate, we define the reward the ratio of unique basic blocks covered by a certain generated program p to the entire test suite I'; that is

$$R(p, I') := B(\mathrm{T}_p) / \bigcup_{\rho \in I'} B(\mathrm{T}_\rho), \tag{5.2}$$

where $B(\mathrm{T}_p)$ is the number of unique basic blocks of the execution trace for a program p and $I' \subset I$ is all the programs generated so far in this test suite. This stepwise reward R is a continuous scalar value that has a range of $(0, 1]$, where 1 is achieved when a specific execution trace covers all the basic blocks that have been tested so far by existing test cases. This reward motivates the mutation steps towards

the training purpose: improve the compiler testing coverage by selecting a critical subsequence inside a seed program and making simple mutations in a trace.

5.3 Learning

To start a deep Q-learning process for compiler fuzzing, we propose FuzzBoost which adopts a constructed forward neural network with two layers connected with non-linear activation functions. We build this end-to-end learning framework with the environment reward calculated based on dynamic trace analysis. In this section, we present the overall learning process for FuzzBoost by illustrating an iteration of fuzz action prediction in the reinforcement learning process for compiler fuzzing as shown in Figure 5.2.

5.3.1 Initialization

We start with an initial input seed $p \in I$, where the choice of p is not constrained but can be any C program even it is not well-formed. We employ the GCC test suite as our sampling pool and randomly selected programs to be our seed inputs. We propose to use a neural network as the Q function to mimic the reasoning for input mutation of compiler fuzzing. This deep neural network maps states (embedding of an extracted substring from seed programs) to estimate Q values for all actions A. Due to the lack of heuristics at the very beginning, we built it a reinforcement learning process, where the neural network is randomly initialized and gradually tunes the parameters θ with learned mutation heuristics calculated with environment rewards.

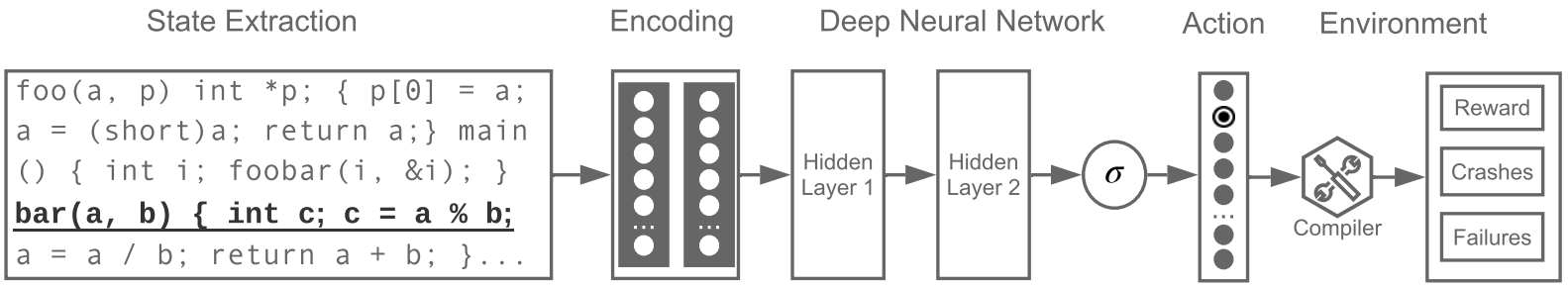


Figure 5.2. Fuzz action prediction in the reinforcement learning process of compiler fuzzing

5.3.2 State Extraction

FuzzBoost will observe a substring within a seed program to predict actions to perform. The substring is extracted from the seed program by customized window, and encoded as $State(p)$. In Section *State*, we defined the states of our Markov decision process to be $I = \Sigma*$. To be more specific, it is a strict substring p' at offset $o \in 0, ..., |p| - |p'|$ and of window size $|p'|$. To make the extracted state controllable, we defined actions in Section *Action*, to shift and resize the window. By performing window-related actions, the fuzzing agent can see the whole program via partially observe fragments consecutively. In other words, FuzzBoost will learn to select the most critical piece of code to mutate incrementally during the training process. After the sequence is extracted, we use word embedding for abstract the sequence into a fixed-sized vector for training.

5.3.3 Deep Q-Network

We implemented the Q-learning module based on Tensorflow [1] 1.14. The deep neural network that used for prediction is a forward neural network with two hidden layers connected with non-linear activation functions. The two hidden layers contain 100 and 512 hidden units respectively, and fully connected with an input layer with 100 units (which is the max window size for input substring) and an output layer with 10 units (which is the size of action space). The goal of the training is to maximize the expected reward. Since the MDP is a finite horizon in our practical design, we adopt a discount rate $\gamma = 0.9$ to address the long-term reward. We set the learning rate $\alpha = 0.001$ to achieve our best-tuned results. We use the decayed epsilon-greedy strategy for exploration in the reinforcement learning iteration, that is the ϵ value was set up to 1 at the very beginning and decays over time until a min value, 0.01 in our configuration, is reached. In this scenario, with the probability

$1 - \epsilon$, the agent selects an action $a = argmax_{a'}Q(x_t, a_t)$, which is the estimated optimal by the on-training neural network. On the contrary, with probability ϵ, the agent explores any other actions with a uniformly distributed choice within the action space $|A|$. To evaluate the proposed framework with the deep Q-Network, we explored its effectiveness under several different initial window sizes. We also explored several non-linear activation functions, including *tanh*, *sigmoid*, *elu*, *softplus*, *softsign*, *relu*. We report experimental results in Section *Evaluation*.

5.3.4 Termination

A mutation episode will terminate when the agent detects a terminal state. In our design, we define three conditions that may trigger the terminal state of mutating of a single seed program: (1) the agent executes the "terminate" action from the neural network prediction; (2) the generated program reaches a maximum number of mutation steps; or (3) the agent generates an invalid action that triggers miscellaneous effects during the reward calculation. The first type of termination will cut the program mutation actively by FuzzBoost while the latter two are passively ended with pre-defined policies. We hard-code the limitation of mutation trace length to be 20 actions in all of our experiments. Theoretically speaking, from the perspective of fuzz testing, the mutation trace can be generated as long as possible to achieve enough randomness. But in practice, to excessively improve the testing efficacy, we set up these policies to enforce our learning agent to learn within the shortest path. Moreover, it is arguable that the length of learning traces is set to be such a small number that during an episode, the agent cannot explore the entire language set. But in our design, all the programs that have achieved a higher coverage will be kept to be a seed and waiting for another round of fuzzing. The methodology applied in our mechanism is the same as conventional coverage-guided fuzzing methods but only has made mutation traces *longer* in one round (compare with 1 step in

Table 5.1. Coverage improvements with different state size

State Size	50	60	70	80	90	100
Coverage Improvement (%)	37.14	36.11	30.29	28.95	28.07	27.94

Table 5.2. Coverage improvements with different activation functions

Activation Function	tanh	sigmoid	elu	softplus	softsign	relu
Coverage Improvement (%)	37.14	28.27	7.48	13.72	14.22	13.26

conventional fuzzing) and *predictable* by a neural network (compare with purely random in conventional fuzzing).

5.4 Experiments

In our research, we proposed a reinforcement learning framework FuzzBoost that incrementally trains a deep neural network to predict mutation actions on a given seed program that improves the compiler testing coverage effectively. We tested the performance of FuzzBoost on a seed input set gathered from the GCC test suites. We randomly sampled 20 C programs in the test suite as our benchmark. We evaluated FuzzBoost in terms of the testing efficacy and compared with a baseline random fuzzing method. We also successfully boosted our fuzzing process by using a pre-trained model for generating new seed programs. All measurements were performed on i7-7700T 2.90Ghz with 12GB of RAM.

5.4.1 Testing Efficacy

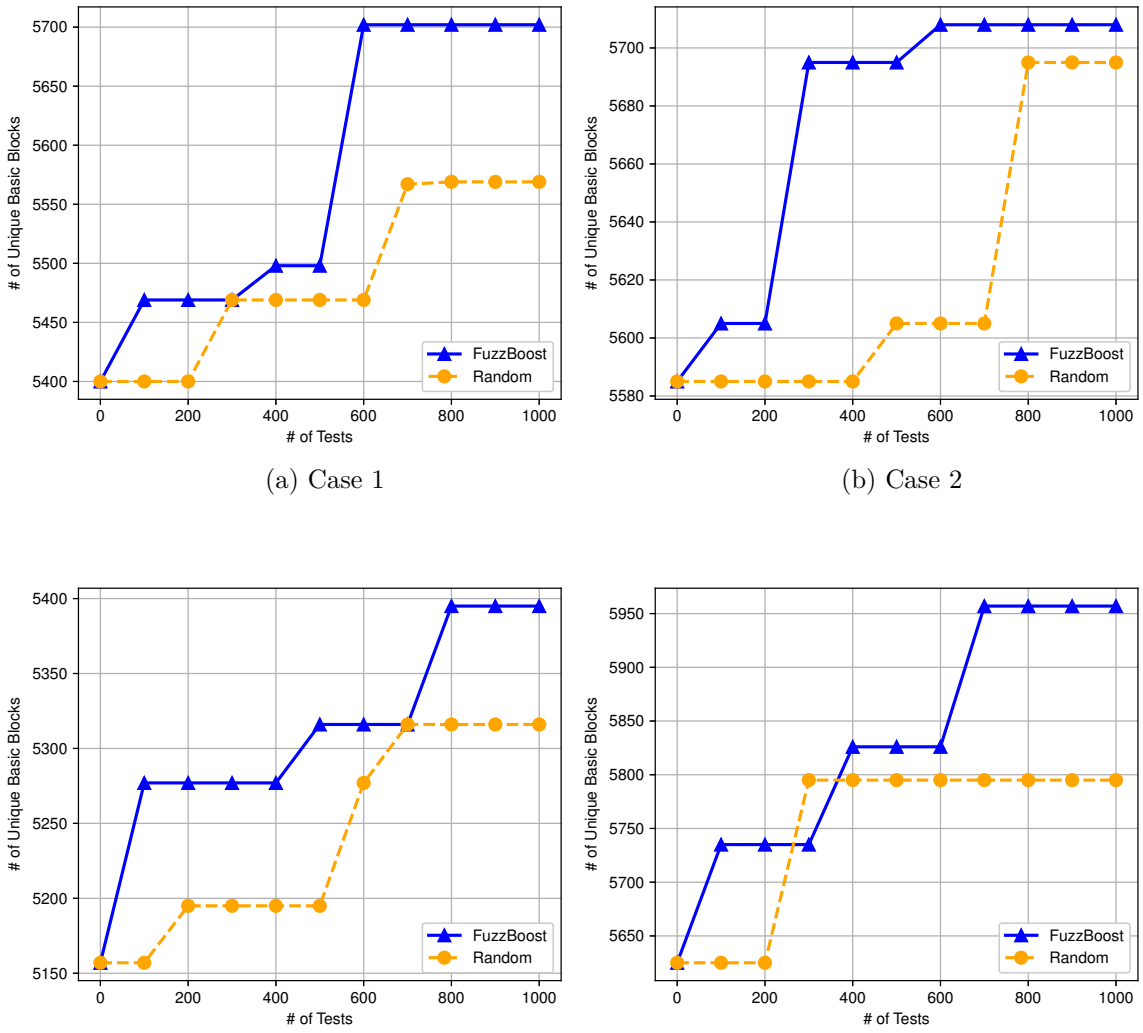
Coverage improvement is the most important measurement for testing. It denotes the overall lines/branches/paths in the original code is being visited. In our design, to improve the performance in this end-to-end learning process, we use an approximation to describe the coverage information, that is the accumulated number of unique basic blocks being executed with the generated new test cases. To show that

Table 5.3. FuzzBoost v.s. random fuzzing

	FuzzBoost	**Random**
Case 1	5,702	5,569
Case 2	5,708	5,695
Case 3	5,395	5,316
Case 4	5,957	5,795

FuzzBoost learning algorithm learns to perform high-reward actions given a seed input observation, we compare the improved testing efficacy against a baseline random action selection policy. The choice of the baseline method uniformly distributed among the action space A and we terminate the actions with the same methodologies as our method described in Section *Termination*. Random mutation is widely-used in software fuzzing tools [101] which is proven to be effective while a good heuristic, such as coverage-guided, is designed. In our framework, we adopt the same methodology to keep or throw away newly-generated programs from our seed test suites. Therefore, we only need to compare the improvement of testing efficacy in one round. We randomly sampled 20 C programs in the GCC test suite, specifically, from the *gcc.c-torture* repository.

Baseline: We performed the experiments with the two different action selection strategies using each of the programs from the sampling pool as the seed. We generated 1,000 new tests from both strategies from the seeds and recorded the accumulated number of unique basic blocks along the execution trace. In the most related work called Deep Reinforcement Fuzzing, for evaluation, 500 pieces are generated for measuring the code coverage improvement. In our experiment, we measured more than 1,000 but we found the improvement does not grow too much after the first 1,000. And to compare with mutation-based fuzzing, we did not use time-based evaluation because our tool has no better performance in terms of scalability because our model will be retrained.



(a) Case 1

(b) Case 2

(c) Case 3

(d) Case 4

Figure 5.3. Compare FuzzBoost with random fuzzing on testing coverage

Compared with the baseline method, in terms of the number of accumulated unique basic blocks as our testing coverage, FuzzBoost achieved testing coverage by 37.14% more on average for the 20 benchmark problems. Figure 5.3 and Table 5.3 demonstrate the coverage improvement of comparisons baseline method and FuzzBoost on mutating four different seed programs, among which the most and least

improvements, 79.17% (case 1, seed1.c) and 12.24% (case 2, seed2.c) respectively, are achieved. From another perspective, an improvement of 5.59% coverage improvement is achieved at most with the newly generated 1,000 programs by FuzzBoost for a single seed.

In similar research done by Rishabh Singh et al. [9] on reinforcement learning the mutation for one single PDF file, it achieves an improvement of approximately 7% improvement on code coverage. Our tool reports 5.6% improvement. The difference may result from different applications where PDF files are easier to be correctly generated and the mutations can be more flexible.

Window Size: Since the size of each seed program varies, and, arguably, the limited number of window size and mutation trace in one episode may put a constraint on exploring the entire seed program. That is, after one episode, the seed program is neither thoroughly observed nor mutated accordingly. Therefore, we analyzed the effectiveness of the current framework with different window size. We increased the initial window size $w = |x'|$ from 50 characters to 100 characters and measured the average reward improvement compare with the baseline strategy on seed1.c.

Table 5.1 shows the results for this experiment. We can see a decreasing improvement when increasing the initial state size. To interpret this result, smaller substrings are better processed than larger ones. In other words, our model learns the best move of small windows and will select the best action accordingly to improve coverage. In addition, the exploration of entire programs is not the key for fuzz testing but making control-flow changes within a limited observations will boost the fuzzing process.

Activation Function: To get a best-tuned model, we are also interested in testing efficacy improvements when applying different activation functions in the

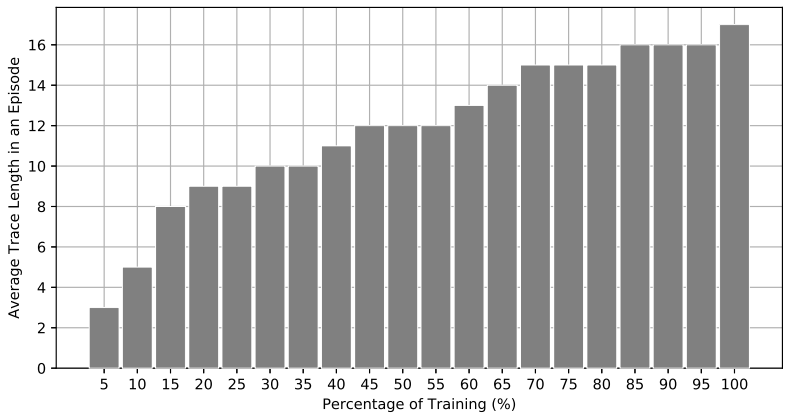
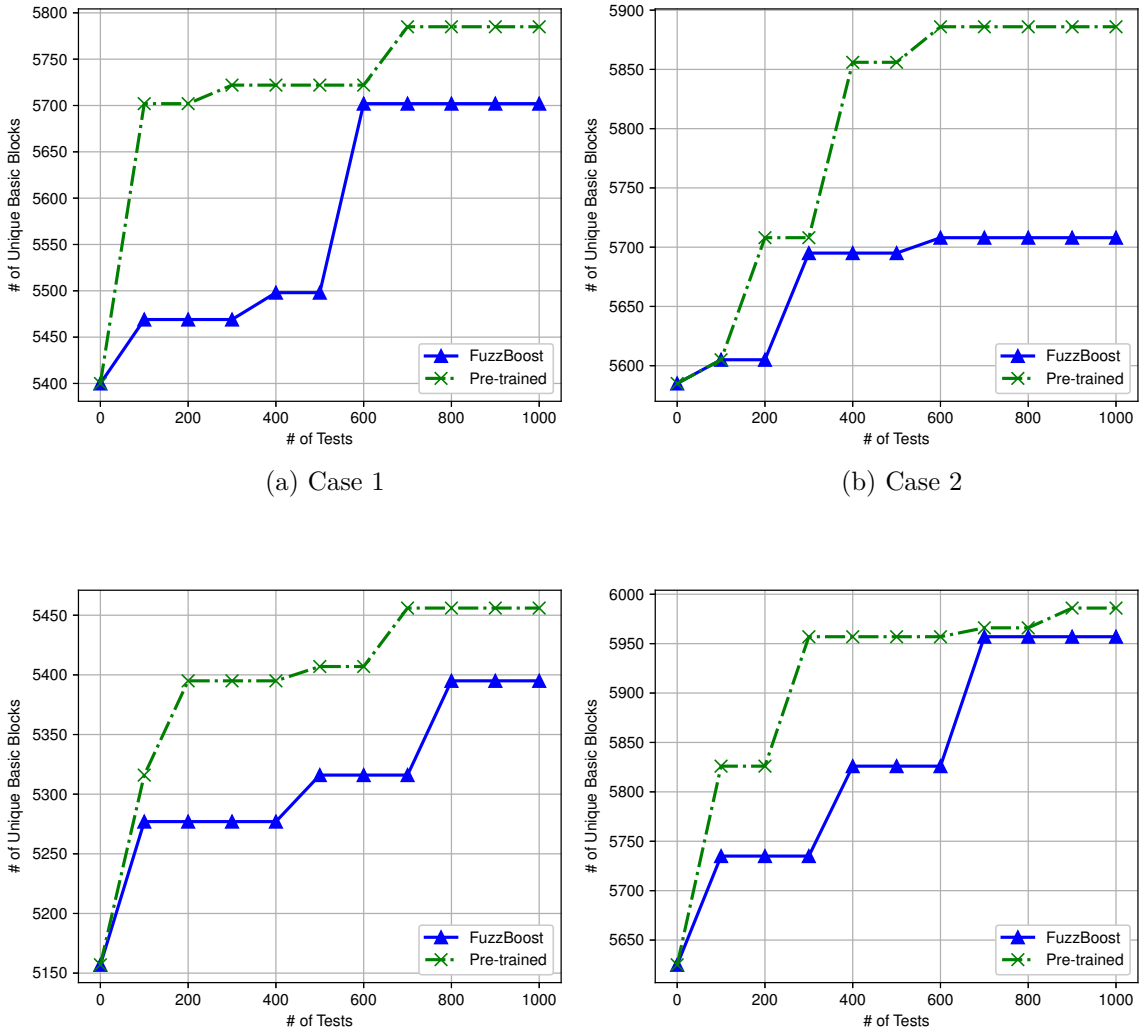


Figure 5.4. Mutation length during training

neural network. We conducted experiments to generate 1,000 new program upon `seed1.c` with FuzzBoost trained with models using different activation functions. Table 5.2 compares the different activation functions with respect to improvement of coverage. All these experiments are done with the Window size 50. We noticed that there is a large variation across different activation functions. For all activation functions provided by the Tensorflow framework, we found the *tanh* function to yield the best result for our setting.

End State: We define the compiler fuzzing as a multi-step decision-making problem and set up the end-to-end learning framework. Theoretically speaking, not like the problem of Go, the end state of FuzzBoost is not deterministic in all cases. In our design, we hard-coded a limit on the length of mutation traces for experiments, but naturally, the traces can be endless to gain enough randomness and achieve a higher reward. Although, to learn an endless mutation trace will improve the testing coverage better than a limited length, to make the two equivalent, we (1) set a limit on mutation times in one episode of training, and (2) if the newly generated code has new Basic Blocks tested, it will be kept in our seed set for another



(a) Case 1

(b) Case 2

(c) Case 3

(d) Case 4

Figure 5.5. FuzzBoost with/without pre-trained model on testing coverage

round of mutating and learning. Therefore, our tool is similar to AFL but has more mutations in one round. In addition, we provided the learning agent an action to actively terminate the episode which varies across the learning stage. Thus, to analyze how end-state evolves, we recorded the distribution of mutation trace lengths under different training stages. Figure 5.4 shows the trace length distributions along

the learning process. From the result, we can see that, with the training goes on, mutation trace lengths are increasing. That is to say, the reward expectation of each mutation action is positive in a well-trained model and it tends to maximize the mutation opportunities in one episode.

5.4.2 Mutation Example

To show how effective FuzzBoost can achieve in program mutations for compiler fuzzing, we show the topmost utilized mutations in the following example. We show an orginal seed (on the left) and its corresponding new generations after mutations (on the right). We discuss each of these abstracted edits that contain a trace of atomic mutations, which explains what is learned by the model during the reinforcement learning process. These mutations are not done within one episode, we just use this one example to show what are the most used mutations and how they look like.

```
1  foo (a, p)
2    int *p;
3  { p[0] = a;
4   a = (short) a;
5   return a;
6  }
7  main () {
8   int i;
9
10  foobar (i, &i);
11
12
13 }
14 foobar (a, b) {
15  int c;
16  c = a % b;
17  a = a / b;
18  return a + b;
19 }
```

Listing 5.1. Original

```
1  foo (a, p)
2    int *p;
3  { p[0] = foobar(a,p);
4   p = (short) a;
5   return a;
6  }
7  main () {
8   int i;
9   for (int a=8; a>0; a--) {
10  foobar (i, &i);
11  }
12  foobar(i, &i);
13 }
14 foobar (a, b) {
15  int c;
16  c = a % b;
17  a = c / b;
18  return a + b;
19 }
```

Listing 5.2. Mutated

Example: By observing the results, we find ① the top most chosen mutation is *insertion*. Usually, the fuzzing engine tries to insert statements with keywords that

do not exist in the original seed file. As shown in *line 9* to *line 11* in the *mutated file*, that the fuzzing engine tries to insert a `for` statement into the seed file. By inserting these non-existing tokens, the compiler should execute the lexical analysis in a way that has not used before. ② the second chosen mutation is *replication* that the fuzzing engine tries to replicate statements locally as shown in *line 12* in the *mutated file*. The replication will trigger the compiler to optimize code which will improve the testing coverage. ③ the third chosen mutation is *replacement* that can be to replace a variable (`a`) with a function call (`foobar(a,p)`) as in *line 3* or to replace a variable (`a`) with another existing variable (`p`). The replacement either makes the statement to be more complex to parse or cause exception handlings such as typecast, or change the control-flow of the seed file, all of which will make the compilation different from the original paths, thereby increasing the testing coverage.

5.4.3 Boosting with pre-training

We next address the question, given an agent which is pre-trained on seed programs $P_{train} = p_i \sim P$, will it improve the testing efficacy faster than learning from scratch? We prepared the training and testing data as follows. We reused the four seed programs in previous experiments which covers the most and least coverage improvement from the initial 20 seed program and created another 9 α-equivalent programs for each seed respectively. We call a program P' is an α-equivalent program of program P when we only perform bound variable renaming on P. We used 80% of them serves as P_{train} and the rest 20% are used for P_{test}. After pre-training the agent on P_{train} for 50 epochs, we saved the model and reused it on P_{test}. It continued the trial-and-error reinforcement learning.

Figure 5.5 shows average coverage improvement using FuzzBoost with an initially arbitrary model and another pre-trained model respectively. We may see that the coverage improvement for the latter case improves drastically towards the

	FuzzBoost	Pre-trained
Case 1	5,702	5,785
Case 2	5,708	5,886
Case 3	5,395	5,456
Case 4	5,957	5,986

Table 5.4. FuzzBoost v.s. pre-trained

highest value in the former case despite the minor difference in the language of two seed programs. In addition, with the training goes on, the coverage was again improved to a new highest value that outperformed previous testing efficacy. It reveals the transferability of a trained model in the context of compiler fuzzing.

5.5 Limitation

To compare with related works is critical but we find it difficult to perform apple-to-apple comparisons. Generation-based fuzzing tools, DeepSmith and Learn&Fuzz [28], generate new programs other than mutating seed programs. Our tool is mutation-based. We rely on seed programs to achieve whole-program validity. Even if our tool has a better performance in terms of testing coverage improvement with a same amount of code synthesized, (it should be according to our observation), it is not convincing to claim that our tool is better. Our tool will generate compilable code for almost all the time, but DeepSmith [19] and Learn&Fuzz do not. There are many code generated from them, rejected at an early stage in the compilation. Therefore the testing procedure is quite shallow. And for AFL [101] and NEUZZ [80], they are efficient for many applications, but not for compilers. On the other hand, AFL is fast; each new file only needs one-step mutation. Although our tool is more efficient in generating more valid and efficient tests, it is not fair to compare with AFL in this way. For NEUZZ, it is grey-box fuzzing that relies on the coverage analysis on target applications. But for compiler testing, the computation cost for code edge

coverage is very high, and that is one of the reasons for using # Basic Blocks tested as an approximation. Thus we cannot directly compare our tool with NEUZZ.

For the reward function, our basic assumption is that, on a single seed program, with limited numbers of mutations, the compiler's coverage has an upper bound, and the exploration towards this upper bound will expand the code coverage. In other words, our tool can be considered as a variant of AFL (or compared with AFL) with a longer trace of mutations in one step. A newly generated code piece, according to the defined heuristics, whether testing coverage is improved, will be kept or deleted from the seed set.

5.6 Summary

In this study, we proposed FuzzBoost, a deep reinforcement learning framework to fuzz off-the-shelf compilers by generating new programs with coverage-guided dynamics. Our proposed end-to-end learning framework that learns to select the best actions to perform automatically without any supervision. It improved the testing coverage on a seed set from the GCC test suite and outperformed the baseline fuzzing agent with a random selection strategy. Moreover, after being pre-trained, it can generalize the strategy to new instances much faster than starting from scratch.

Chapter 6
Discussion

We presented three studies that automatically generate small pieces of code for compiler testing. In this chapter, we describe the rationale of the three sub-projects and compare them horizontally in terms of (1) models and (2) improvement of testing coverage.

6.1 Research Rationale

In this section, we describe the rationale for proposing each of these three sub-projects and the challenges that we have faced.

6.1.1 DeepFuzz

In the first study, DeepFuzz, we proposed to use a *Sequence-to-Sequence* model to learn a language model ovecr a set of existing code. This work is the *first* to automatically generate C programs by insertion mutations and with which, the testing coverage of GCC and LLVM was improved in a higher efficiency compared with human-crafted generation rules. However, DeepFuzz is still based on training data. In other words, the newly-generated programs are like new combinations of existing code pieces. Although it brings in code diversity while maintains the code validity, both syntactically and semantically, there are still rooms to improve.

6.1.2 AlphaProg

In the second stage, where we proposed to train a neural-network-based language model, for continuously generating new programs, with *no* training data. The good thing about no training data is that, there are chances to generate non-existing language patterns step-by-step. Theoretically speaking, if we use a well-designed heuristic to drive the synthesis process, the exploration space will be larger than to purely train a model based on existing code pieces. On the other hand, if there is no training data, the efficiency of the training process will be demotivated. That is to say, the positive samples will be very sparse, which is true in our analysis, especially when we set up the hyper-parameter to favor exploration. We adopted the reinforcement learning method for training this neural network-based language model gradually. However, there are a few challenges to overcome to fit into our problem:

- **Inefficiency:** This was the first challenge we encountered when we first made a trial to synthesis C program from scratch, character by character. This is because there were too few positive samples to efficiently train a model. Humans can be taught how to say a correct sentence with one training example, which is not the case with computers. The observation revealed a low validity rate (less than 1%) even though we set up an experiment of growing a fixed-length (20 characters) short program (the search space is 128^20). Therefore, we simplified the problem by introducing a new scenario, applying the method to a programming language called BF instead of C. There are only 8 characters in this new language, thus decreasing the search space exponentially.

- **Reward function:** It is certainly true that defining a reward function is difficult, especially in our task. In a pure program synthesis task, to synthesize a

correct (syntactically/semantically) program is the goal. But in our scenario, we would like the synthesized program to be both valid (syntactically/semantically correct) so it can pass the compiler checks, as well as diverse so it can trigger different execution paths when the compiler attempts to compile it. Before the presented experiments in Chapter 4, we had a few failures. Our first trial was to merely use program validity as the reward. The final observation showed that the model converged in a fast way and fell into the local optimal despite how much dropout or randomness we set. Our second trial was designed to combine validity with program diversity. We had the experience of combining a different number of characters into the reward function. But the model will converge fast and fell into the local optimal again. The difference is that the model stopped at synthesizing a valid and diverse (contains 8/8 characters) program. Then, we realized that the program diversity cannot be calculated statically, instead, we stored existing programs to calculate this *diversity*. Essentially, the synthesized programs are for compiler testing, and we only later found that a better way to evaluate this diversity is to use testing coverage, and an approximation is to count the newly executed basic blocks on the execution trace. We designed our experiments based on all of this pre-knowledge, which eventually showed us a best-tuned model for generating BF programs from scratch.

- **Local optimal:** This problem is common across all machine learning tasks, but the problem is well defined and well-studied. One solution is to use dropout method or just add more randomness. In reinforcement learning tasks, to add randomness is realized by tuning the hyper-parameter, ϵ. We first tried to use a fixed-value for ϵ, but results revealed either a lack of exploration (if ϵ is too small) or difficulty in converging where the total regret is large (if ϵ is too large).

Therefore, we finally adopted a decaying ϵ strategy that included changing the value of ϵ. Again, there is a trade-off between exploitation and exploration while we are talking about picking a decay schedule for ϵ, i.e. the gap. It is difficult to determine this value because the more uncertainty associated with an action value, the more important it is to explore that action. Therefore, we tuned this hyper-parameter based on an evaluation of cumulative average reward. We also found that there are interactions between this gap and learning rate as well. In this case, we batch updated these two values to find an optimal model.

6.1.3 FuzzBoost

We then came to the third stage, where we proposed a neural network-based mutation method to improve compiler testing coverage starting from a single seed program. As described in Section 6.1.2, we faced the challenge from the large search space when we considered the problem of generating C programs from scratch. Another solution for this issue is to base the generation on seed files. In other words, we can train a neural network to mutate seed programs that share the same task as a well-known fuzzing tool called AFL. However, to conduct this study, there are still challenges other than search space:

- **Mutation rules:** Since we are trying to implement a deep neural network to select the best action to mutate an existing code piece, there should be a few mutation rules to choose from. We hope the local mutation will improve corresponding testing coverage while keeping the validity of original seed programs. We could start from the mutation methods like AFL, which works on binaries and does simple bit replacements or flips, though it overlooks the complexity of C language. The synthesized programs rarely pass the

lexical/semantic checks of compilers, and this hinders its efficiency with respect to improving the testing coverage. Therefore, we defined a set of token-level mutations that conforms C language production rules, including *insert* a token, *switch* two or more tokens, *replace* a token, or *change* the window size or offset to enable another substring to observe and mutate. Since the insertion set contains the complete C language keywords, digits and characters, the language set after mutation theoretically equals to the C language set.

- **Reward function:** Similar to AlphaProg, the synthesized programs are used for compiler testing, and it is critical to calculate the reward function based on testing coverage. In the AlphaProg work, better coverage improvement was achieved by combining program validity and testing coverage. In contrast to AlphaProg, where we need the neural network to learn how to generate syntactically correct programs, we did not include the validity as part of the reward calculation, which is because the program validity will be maintained through the mutation rules, and the learning efficiency can thereby be improved without such calculation.

6.2 Comparation

6.2.1 Model

In our first two studies, DeepFuzz and AlphaProg, we adopted the *Sequence-to-Sequence* model to encode the language pattern. However, in the third study, FuzzBoost, we adopted a two-layer fully connected dense network. The reason we choose different models for the three projects is due to the differences in the synthesis task. For DeepFuzz and AlphaProg, the neural network serves as a language model which should be responsible for sentence embedding as well as decoding. Each of the synthesized characters is correlated in the training sequence.

However, for FuzzBoost, the neural network serves as an action selector which bases its knowledge of the given sentence embedding to select a mutation action. The actions in the mutation sequence are independent in our scenario. Our tool can be considered as a variant of AFL with an extended trace of mutations in one step. Moreover, a newly generated code piece, according to the defined heuristics, whether testing coverage is improved, will be kept or deleted from the seed set.

6.2.2 Coverage Improvement

Since our objective was to synthesize programs for compiler testing, the most important measurement of performance we care about is the testing coverage. Although the three tasks share the same goal, they are slightly different from each other. For DeepFuzz, our main objective is to use insertion mutation to combine learned language patterns into seed programs from a test suite. In other words, the neural network encodes language patterns and the program diversity is improved by our mutation strategy. Essentially, the testing coverage is improved because the control-flow diversities of original seed files are improved. Additionally, we have to claim that the final coverage of GCC (82.27%) is achieved because we choose to mutate based on the original test suite (75.13%). However, in contrast to DeepFuzz, AlphaProg synthesizes programs from scratch. And the neural network of AlphaProg is responsible for synthesizing both valid and diverse programs. As a result, the first 10,000 programs achieved 84,500 basic blocks and the second 10,000 programs achieved 110,500 basic blocks of the target compiler. If we treat the coverage of the first 10,000 programs as the baseline, DeepFuzz achieved 7.14% improvement while AlphaProg achieved over 30%. Although it is arguable to conclude higher efficiency of reinforcement learning than based on training data, the performance of reinforcement learning is acceptable even though no training set is provided. For FuzzBoost, our main objective is to improve the testing

coverage based on one single seed program. Therefore, we cannot directly compare FuzzBoost with the other two.

Chapter 7
Conclusion

Compiler testing is critical for assuring the fundamental correctness of computing systems. Fuzzing is one of the most common mainstream technologies used to assist with compiler testing. In our first study, we proposed an automatic grammar-based fuzzing tool called DeepFuzz that learns a generative recurrent neural network that continuously provides syntactically correct C programs to fuzz off-the-shelf compilers, GCC and Clang. We conducted a detailed study on analyzing how key factors, i.e., sampling method and generation strategy, effect on the accuracy of this generative model, and how different improvements of testing efficacy are achieved. DeepFuzz generated 82.63% syntax valid programs and improved the testing efficacy with respect to line, function and branch coverage. With the preliminary evaluation, we found and reported 8 bugs in GCC, all of which have been actively addressed by developers.

In our second study, we proposed a reinforcement learning-based approach to continuously generate BF programs for BF compiler fuzzing. With no training data set required, the model was initialized with random weights at the very beginning and it evolved with environment rewards provided by the target compiler we are going to test. During the performance of the learning iterations, the neural network model gradually learns how to write valid and diverse programs to improve testing efficacies under the three different reward functions we defined. We incorporated

the proposed method into a prototyping tool called AlphaProg. We detailed the configuration of our model and open-sourced the code. Our study revealed the overall effectiveness of AlphaProg for compiler testing. We also compared metrics under the different reward functions and explained the improved testing coverage by analyzing the generated programs. Our tool helped to find two important bugs of a production BF compiler, BFC, and all were confirmed and well-addressed by the project owner.

In our third study, we proposed FuzzBoost, a deep reinforcement learning framework was used to fuzz off-the-shelf compilers by generating new programs with coverage-guided dynamics. Our proposed end-to-end learning framework learns to select the best actions to perform automatically without any supervision. It improved the testing coverage on a seed set from the GCC test suite and outperformed the baseline fuzzing agent with a random selection strategy. Moreover, after being pre-trained, it was able to generalize the strategy to new instances much faster than starting from scratch.

As part of further studies, we suggest a more in-depth analysis of the reasons why the proposed deep neural network functions in this scenario, as well as determining what syntax patterns of C are encoded in neural networks and what is lost. Our preliminary study on using the LSTM network for program synthesis has achieved good performance in terms of pass rate, which is the ratio of syntactically correct programs to the entire programs being synthesized. Therefore, we suggest investigating whether a neural network trained for program synthesis learns syntactic information on the source side as a by-product of training.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013.

[3] David Alvarez-Melis and Tommi Jaakkola. A causal framework for explaining the predictions of black-box sequence-to-sequence models. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

[4] Saul Amarel. *Representations and modeling in problems of program formation.* Rutgers University. Livingstone College. Department of Computer Science, 1970.

[5] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of the 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564. IEEE Publ. Piscataway, NJ, 1995.

[6] Sahil Bhatia and Rishabh Singh. Automated Correction for Syntax Errors in Programming Assignments Using Recurrent Neural Networks. *arXiv preprint arXiv:1603.06129*, 2016.

[7] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3), 1983.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[9] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122. IEEE, 2018.

[10] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[11] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 180–190. IEEE, 2016.

[12] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 197–208, New York, NY, USA, 2013. ACM.

[13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 2014.

[14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[15] Clang: a C language family frontend for LLVM. clang.llvm.org, 2018.

[16] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[17] ISO Committee. SC22/WG14. ISO/IEC 9899: 2011. *Information technology, Programming languages, C*, 2011.

[18] 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*, 2018.

[19] 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*, 2018.

[20] Baptiste David. How a simple bug in ml compiler could be exploited for backdoors? *arXiv preprint arXiv:1811.10851*, 2018.

[21] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[22] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[23] Amir Elmishali, Roni Stern, and Meir Kalech. Data-augmented software diagnosis. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016.

[24] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[25] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

[26] GCC, the GNU Compiler Collection. gcc.gnu.org, 2018.

[27] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[28] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[29] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

[30] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, and John Agapiou. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

[31] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[32] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

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

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[35] Wilfred Hughes. BFC: An industrial-grade brainfuck compiler, 2019.

[36] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. RedDroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–199. IEEE, 2018.

[37] Yufei Jiang, Xiao Liu, Fangxiao Liu, Dinghao Wu, and Chimay J Anumba. An analysis of BIM web service requirements and design to support energy efficient building lifecycle. *Buildings*, 6(2):20, 2016.

[38] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems*, pages 190–198, 2015.

[39] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 782–792. ACM, 2015.

[40] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[41] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Combining stochastic grammars and genetic programming for coverage testing at the system level. In *International Symposium on Search Based Software Engineering*, pages 138–152. Springer, 2014.

[42] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. OpenNMT: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*, 2017.

[43] A. S. Kossatchev and M. A. Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, Jan 2005.

[44] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[45] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1945–1954. JMLR. org, 2017.

[46] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[47] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[48] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[49] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[50] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. In *ERTS 2016: 8th European Congress on Embedded Real Time Software and Systems*, 2016.

[51] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2), 2009.

[52] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

[53] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[54] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[55] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. 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, 2017.

[56] Xiao Liu. BFC hangs due to compile-time evaluation (issue), 2019.

[57] Xiao Liu, Brett Holden, and Dinghao Wu. Automated synthesis of access control lists. In *2017 International Conference on Software Security and Assurance (ICSSA)*, pages 104–109. IEEE, 2017.

[58] Xiao Liu, Yufei Jiang, and Dinghao Wu. A lightweight framework for regular expression verification. In *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, pages 1–8. IEEE, 2019.

[59] Xiao Liu, Yufei Jiang, Lawrence Wu, and Dinghao Wu. Natural shell: An assistant for end-user scripting. *International Journal of People-Oriented Programming (IJPOP)*, 5(1):1–18, 2016.

[60] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

[61] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. Automatic grading of programming assignments: an approach based on formal semantics. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 126–137. IEEE, 2019.

[62] Xiao Liu and Dinghao Wu. PiE: programming in Eliza. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 695–700. ACM, 2014.

[63] Xiao Liu and Dinghao Wu. A lightweight framework for regex verification. *ACM Graduate Student Research Competition at PLDI*, 2017.

[64] Xiao Liu and Dinghao Wu. From natural language to programming language. In *Innovative methods, user-friendly tools, coding, and design approaches in people-oriented programming*, pages 110–130. IGI Global, 2018.

[65] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, USA, 2005. ACM.

[66] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov), 2008.

[67] Andrei Andreevich Markov. The theory of algorithms. *Trudy Matematicheskogo Instituta Imeni VA Steklova*, 42:3–375, 1954.

[68] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 2007.

[69] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[70] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, and Georg Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[71] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

[72] Aniruddh Nath and Pedro M Domingos. Learning tractable probabilistic models for fault localization. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016.

[73] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[74] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[75] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[76] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998.

[77] Brian Raiter. Brainfuck: An eight-instruction turing-complete programming language, 1993.

[78] Michael Rash. A collection of vulnerabilities discovered by the afl fuzzer (afl-fuzz), 2019.

[79] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA, 2012. ACM.

[80] Dongdong Shi and Kexin Pei. NEUZZ: Efficient fuzzing with neural program smoothing. *IEEE Security & Privacy*, 2019.

[81] Xing Shi, Inkit Padhi, and Kevin Knight. Does string-based neural MT learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.

[82] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[83] Phillip D Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

[84] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In *ISSTA*, 2016.

[85] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning*, 2011.

[86] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

[87] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 1998.

[88] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[89] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *arXiv preprint arXiv:1804.02477*, 2018.

[90] Richard J Waldinger and Richard CT Lee. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252. Morgan Kaufmann Publishers Inc., 1969.

[91] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[92] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 61–64. ACM, 2018.

[93] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 657–674, 2019.

[94] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 235–252, 2017.

[95] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[96] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute, 1996.

[97] Jingbo Yan, Yuqing Zhang, and Dingning Yang. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks*, 6(11):1319–1330, 2013.

[98] 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*, 2011.

[99] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[100] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: Fuzzing without valid seed inputs. In *Proceedings of the 41st International Conference on Software Engineering, ICSE*, volume 2019, 2019.

[101] Michal Zalewski. American fuzzy lop, 2014.

[102] Tom Zamir, Roni Tzvi Stern, and Meir Kalech. Using model-based diagnosis to improve software testing. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-14)*, 2014.

[103] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 347–361, New York, NY, USA, 2017. ACM.

Vita

Xiao Liu

Xiao Liu finished her M.S. and Ph.D. at the College of Information Sciences and Technology, The Pennsylvania State University, in 2015 and 2019, respectively. She was advised by Dr. Dinghao Wu. Her research interest subsumes software engineering and program synthesis. Before she went to Penn State, she received her bachelor's degree from Nanjing University of Posts and Telecommunications in 2013 in Communication Engineering. During her Ph.D. study, she has published research papers [36, 37, 57, 58, 59, 60, 61, 62, 63, 64, 93, 94] in cybersecurity and software engineering conferences.