Title

Greybox Fuzzing and Its Applications

Author

Rong, Yuyang

Publication Date

2024

Peer reviewed|Thesis/dissertation

Greybox Fuzzing and Its Applications

By

YUYANG (PETER) RONG
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Hao Chen, Chair

Matt Bishop

Mohammad Sadoghi

Committee in Charge

2024

Contents

Abstract xvii

Acknowledgments xix

1 Introduction **1**

1.1 Greybox fuzzing 1

1.2 Improving fuzzing performance using principled techniques 3

1.3 Specialized fuzzing for the LLVM backend 5

1.4 Enhancing program understanding using fuzzer generated test cases 6

2 Integrity: finding integer errors by targeted fuzzing **8**

2.1 Introduction 8

2.2 Design 11

2.2.1 Exploitation 11

2.2.2 Exploration 14

2.3 Implementation 16

2.4 Evaluation 17

2.4.1 Juliet test suite 17

2.4.2 Real world applications 20

2.4.3 Which non-crashing error is harmful? 22

2.4.4 Comparison with Angora + UBSan 27

2.4.5 Instrumentation reduction 27

2.5 Related work . . . 28
2.5.1 Detecting integer overflow . . . 28
2.5.2 Coverage-directed fuzzers . . . 29
2.5.3 Bug-directed fuzzers . . . 29

3 Valkyrie: improving fuzzing performance using principled techniques 31

3.1 Introduction . . . 31
3.2 Background and motivation . . . 37
3.3 Design . . . 39
3.3.1 Collision-free context-sensitive branch counting . . . 39
3.3.2 Compensated mutation assisted solver . . . 44
3.3.3 Proactive bug exploitation . . . 54
3.4 Evaluation . . . 56
3.4.1 Magma benchmark . . . 58
3.4.2 Real-world open-source programs . . . 62
3.4.3 Effectiveness of deterministic branch counting . . . 64
3.4.4 Effectiveness of deterministic solver . . . 66
3.4.5 Bug finding ability of Valkyrie . . . 68
3.4.6 Summary . . . 70
3.5 Discussion . . . 70
3.5.1 Unsolved predicates . . . 70
3.5.2 Bug detection . . . 71
3.5.3 Branch counting effectiveness . . . 71
3.6 Related work . . . 71
3.6.1 Branch counting methods . . . 71
3.6.2 Predicate Solving Methods . . . 72
3.6.3 Targeted fuzzers . . . 72

3.6.4 Machine learning based fuzzers . . . 73

4 IRFuzzer: specialized fuzzing for LLVM backend code generation 74

4.1 Introduction . . . 74

4.2 Background . . . 78

4.2.1 LLVM . . . 78

4.2.2 Coverage guided fuzzing . . . 81

4.2.3 Challenges in compiler fuzzing . . . 82

4.3 Design . . . 84

4.3.1 LLVM IR mutation . . . 85

4.3.2 Matcher table feedback . . . 90

4.4 Implementation . . . 92

4.5 Evaluation . . . 92

4.5.1 Baseline comparison . . . 94

4.5.2 Comparison with end-to-end fuzzers . . . 95

4.5.3 Individual contributions . . . 97

4.5.4 Bug categories and analysis . . . 98

4.5.5 Bugs case study . . . 100

4.6 Related work . . . 103

5 Understanding programs by exploiting fuzzer generated test cases 105

5.1 Introduction . . . 105

5.2 Related work . . . 106

5.2.1 Code representation learning . . . 107

5.2.2 Fuzzing . . . 107

5.3 Method . . . 108

5.3.1 Fuzzing for obtaining inputs and outputs . . . 108

5.3.2 Model . . . 111

5.3.3 Prompting . . . 111
5.4 Experimental results . . . 112
5.4.1 Clone detection results . . . 114
5.4.2 Code classification results . . . 116
5.4.3 Ablation study . . . 117
5.4.4 Data scale . . . 118
5.4.5 Case study . . . 119
5.5 Limitations . . . 121

6 Code representation pre-training with complements from fuzzer generated test cases 123

6.1 Introduction . . . 123
6.2 Related work . . . 126
6.3 Code representation pre-training . . . 128
6.3.1 Fuzzing code corpus . . . 129
6.3.2 Static and dynamic information modeling . . . 130
6.3.3 Model training and inference . . . 132
6.4 Experiments . . . 133
6.4.1 Code representation learning . . . 134
6.4.2 Ablation study . . . 140
6.5 Future work and limitations . . . 142

7 Conclusion 144

7.1 Improving fuzzing performance using principled techniques . . . 144
7.2 Specialized fuzzing for the LLVM backend . . . 145
7.3 Enhancing program understanding using fuzzer generated test cases . . . 146
7.4 Future work . . . 147

List of Figures

1.1 The workflow of greybox fuzzers. 2

2.1 Cumulative distribution function (CDF) of the average L^1 distance (Equation 2.4.1) between the output of two decoders on the same input JPEG image. The CDF of the normal images is cleanly separable from that of the exploit images. 26

3.1 Examples of branches that do not require instrumentation. Only thickened edges need instrumenting. 45

3.2 Arithmetic mean of number of integer and memory bugs triggered per trial per day. The black line shows 95% confidence interval. Valkyrie's performance is the same across ten trials. 58

3.3 Siginificant plot of Valkyrie. Valkyrie is superior than state-of-the-art on libpng, libxml2, and poppler. 60

3.4 Branch coverage of six fuzzers in 24 hours time. Valkyrie-br is Valkyrie with only branch coverage improvement, Valkyrie-solver is Valkyrie with only solver improvement. Both design increased branch coverage compared with Angora in all programs. Overall, Valkyrie ranked #1 on geometric mean number of branches reached. 63

3.5 Branch coverage of four fuzzers in 24 hours time. Valkyrie not only finds more branch coverage but also is the fastest one on eight of ten applications thanks to deterministic algorithms. 64

3.6 Plots of branch coverage over 24 hours for all evaluated fuzzers on *objdump* and *pdftotext*. 64
3.7 The difference between preidcates solved by Valkyrie and Angora over 16 seeds. . . 67

4.1 Overview of IRFuzzer. Green shaded components are the contributions of this chapter, orange shaded components are AFL++, and blue shaded components are from LLVM. We first create a LLVM IR mutator that guarantees the correctness of the generated input (Section 4.3.1). We introduce a new coverage metric to keep track of the backend code generation while providing a mutation guide to the mutation module (Section 4.3.2). 76
4.2 Examples of failed and successful CFG mutations, respectively. 77
4.3 LLVM can be roughly partitioned into three components, frontend, middle end, and backend. 79
4.4 AFL can be modelled as a four-stage loop that tests the executable repeatedly. . . . 80
4.5 An example of how IRFuzzer mutates a module using different strategies. 84
4.6 Distributions of bugs found by IRFuzzer. IRFuzzer has found **78** new bugs, **57** have been fixed. 99
4.7 A piece of code we generated, simplified to CFG only. Two optimization passes are involved in this compiler hang. `TurnSwitchRangeIntoICmp` will transform Figure 4.7a into Figure 4.7b. `FoldValueComparisonIntoPredecessors` will undo the transformation, causing an infinite loop. 100
4.8 Two bugs we found in LLVM codebase. Both of them will lead to compiler crash and have been fixed. 102

5.1 Per-problem clone detection performance on the POJ104 test set, using *CoderBERT+FineT* or *CoderBERT+FuzzT*. The horizontal axis shows the ID of the POJ104 problems, and the vertical axis is the MAP@R. 120

5.2 Per-problem clone detection performance on the POJ104 test set, using *UniXcoder+FineT* or *UniXcoder+FuzzT*. The horizontal axis shows the ID of the POJ104 problems, and the vertical axis is the MAP@R. 121

6.1 An illustration of different pre-training and fine-tuning paradigms for code understanding. 124

6.2 An illustration of implementation variations of the same functional purposes. Figure 6.2a is dramatically different from its Figure 6.2b iterative counterpart regarding their structures, even though their functional equivalence is explicitly demonstrated by the consistent behaviors. On the other hand, the subtle change in Figure 6.2c is barely observable in comparisons to Figure 6.2b, but can lead to distinct execution results. 125

6.3 An overview of FuzzPretrain. **(a)** The input sequences are composed of both the code and test cases which are concatenated then encoded by a transformer. FuzzPretrain learns code feature representations by **(b)** static information modeling (SIM) through masked tokens predictions, **(c)** dynamic information matching (DIM) to match test cases to code, and **(d)** dynamic information distillation (DID) to summarize the holistic information about code structure and functionality. 128

6.4 The fuzzing process collects test cases that embed dynamic behavior from program datasets. 129

6.5 Qualitative studies for code search. The functional equivalence of code snippets are marked by their shared colors. Only a few classes are highlighted with bright colors to be visually distinguishable. 136

6.6 Effects of different components for dynamic information modeling. We constructed three variants of FuzzPretrain with either DIM or DID or both being removed to be compared. 140

6.7 Dynamic information modeling by MLM. The "Mask" variant replaces DIM by MLM for both code and test cases while "Match" is the design we adopted and "Both" is the combination of the two. 140
6.8 Positive pairs in DID. The "Execution" variant constructs the positive pairs in DID using code T^s and its test cases T^d, and our "Holistic" design contrasts code to its concatenation with test cases $T^s \oplus T^d$. 141

List of Tables

2.1 Verified, unique arithmetic errors that Integrity found in real world applications, compared with Angora + UBSan. Note that the total numbers of unique errors at the bottom are fewer than the sums of the rows above because some programs share the same library and therefore we removed these duplicate errors when calculating the totals. 9

2.2 Errors that Integrity found on the Juliet test suite. A "-" cell means that the corresponding test set on the top contains no corresponding subset on the left. Integrity found all the errors with no false positive. Every test contains one inserted arithmetic error except subset s02 of CWE197, where half of its inserted bugs contain two truncation errors each. 19

2.3 Unique errors that Integrity found in common open-source programs. Note that the total numbers of unique errors at the bottom are fewer than the sums of the rows above because when calculating the totals we removed the duplicate errors in the libraries shared by different programs. 21

2.4 Benign arithmetic errors determined by statistics of traces. We use the benign errors found by manual inspection as the ground truth when calculating the precision and recall of the benign errors determined by statistics of traces. 24

2.5 Number of instrumented arithmetic operations before and after instrumentation reduction 28

3.1 Examples of path compressions in Figure 3.1. Grey areas in column five means that we didn't allocate memory for thoese edges. Notice how edge `f` and `h` *must* be executred, thus there is no need to instrument them. 46
3.2 Conversion table between branch predicate expressions, their corresponding objective functions and solver targets. δ represents the smallest possible positive value that the numerical type can represent. For integers, $\delta = 1$. 46
3.3 The list of fuzzers we used in our evaluation. Included are their respective versions and the arguments we provided to invoke the fuzzer. 57
3.4 The list of projects we used in our evaluation. Included are their respective versions, the binary we used and the arguments we provided to invoke the binaries. 57
3.5 Average time used to trigger a bug in Magma. Bolded text shows the fastest to trigger a bug. 59
3.6 Bitmap size for Valkyrie before and after optimization. On average we reduced 69% of all instrumentations and 28% of runtime. 66
3.7 Bitmap utilization for AFL and Angora on open-source programs. We evaluated their respective utilizations under default sizes and adjusted sizes. "*" indicates failure, AFL refuses to run *jhead* with only 8K bitmap. 67
3.8 Bugs found by Valkyrie and Angora. Valkyrie found six bugs in three programs while Angora only found three. 68

4.1 Instruction modeling for IR instructions. 88
4.2 Matcher table size in all architectures in LLVM on commit `860e439f`. Since GlobalIsel is a new CodeGen framework introduced in 2015, only eight architectures have implemented it. 90

4.3 Branch table coverage and matcher table coverage on 29 target CPUs across 12 targets in SelectionDAG. Statistics are the arithmetic mean over five trials. Bold entries are the best among baseline fuzzers. FM means AFL++ coupled with FuzzMutate, IRF means IRFuzzer 94

4.4 Average branch table coverage and matcher table coverage of CSmith (CS), GrayC, and IRFuzzer (IRF). `O2` and `O3` stands for different optimization levels. Bold entries are the winners. 96

5.1 Dataset statistics. 112

5.2 Clone detection results on CodeNet. Compared with normal fine-tuning (FineT), our fuzz tuning (FuzzT) leads to significant improvements and new state-of-the-arts. C++1000* contains 16% of all problems, which is a roughly 6.3x downsample of the original dataset (see Table 5.8 for results on other scales). Bold stats are better. . . 112

5.3 Clone detection results on POJ104. Our fuzz tuning (FuzzT) leads to state-of-the-art results. Bold stats are better. 113

5.4 Code classification results on CodeNet. Our fuzz tuning (FuzzT) leads to new state-of-the-arts. C++1000† contains 40% of all problems, which is a 2.5x downsample of the original dataset (see Table 5.9 for results on other data scales). Bold stats are better. 114

5.5 Code classification results on POJ104. Our fuzz tuning (FuzzT) leads to new state-of-the-arts. Bold stats are better. 114

5.6 Comparing using raw and decoded fuzzing test cases in tuning clone detection (CD) and code classification (CC) models on POJ104. MAP@R and the error rate are evaluated for the two tasks, respectively. Bold stats are better. 117

5.7 Comparing different prompts for our fuzz tuning on the clone detection (CD) and code classification (CC) tasks on POJ104. Bold stats are better. 117

5.8 How different methods scale with the size of training/tuning dataset on the C++1000 *clone detection* task. Bold stats are better. 118
5.9 How different methods scale with the size of training/tuning dataset on the C++1000 *code classification* task. Bold stats are better. 119

6.1 Evaluations on code search. Results of our base models (CodeBERT and UniXcoder) are from [170]'s paper, which are marked in grey because of different training data. The first and second rows in the header indicate the programming language of the query and the target code snippets, respectively. The column "DYN" indicates whether a model was trained using the test cases or not. mAP scores (%) are reported. 135
6.2 Evaluations of code representations on inductive code search. 137
6.3 Evaluations in novel data domains. Results of the base models are marked in grey as training on different data from ours. Results marked with * are reproduced using the checkpoints from authors. 138
6.4 Comparisons with the state-of-the-arts that adopt the same backbone network as ours with 125M parameters. Results marked with * are reproduced using the checkpoints from authors. 139

List of Code Snippets

2.1 A test in CWE197 s02, which contains two truncation errors on Line 4 and Line 6. . 19

2.2 An example of benign integer overflow. After LLVM optimization passes, the C program was translated into the IR shown in the figure, the syntax slighted modified for readability. On Line 3, the `add` instruction overflows when the loop variable `%iter_var` is 0, but the overflown result will never be used. 22

2.3 Divide by zero error in *jmemmgr.c* of *libjpeg-ijg* happens when the parameter `samplesperrow` is zero. 22

3.1 Code snippet copied from libjpeg-9d. The program requires the length to be a specific amount to continue. 39

3.2 Two seemingly easy bugs AAH001 and MAE014 in Magma. Valkyrie can trigger this bug in seconds while other fuzzers can take hours. 61

3.3 Code snippet copied from xpdf. The program accesses the array without checking the bound. 69

4.1 SelectionDAG in LLVM that consumes a matcher table to do instruction selection. . 81

4.2 A piece of LLVM IR program generated by function generation(Section 4.3.1.1). The function returns a 64 bit integer, so we allocate a stack memory and load from it to return. We will fill the memory in later mutations. 84

4.3 IR program mutated from Listing 4.2. Line Line 4 to Line 10 are introduced by *s*CFG insertion(Section 4.3.1.2). We insert *s*CFG by splitting the `Entry` block into two and generate a `switch` instruction. 84

4.4 IR program mutated from Listing 4.3. Instruction insertion(Section 4.3.1.3) generated Line 8, Line 12, and Line 15. The placeholder memory is also used by `%PHI` to avoid undefined behavior (Line 16). 84
4.5 A snippet of code in AArch64 where the index (`IntImm`) is not sanitized before usage. This diff is our patch to fix this bug. 102
4.6 A snippet of code in LLVM where index of a vector is treated as signed value. This diff is our patch to fix this bug. 102

Abstract

Reliable software is vital to society. Much effort has been spent to ensure the robustness and reliability of the software, including unit testing, model checking, static analysis, etc. However, these approaches do not scale well.

Greybox fuzzing can test the software with little or no human intervention. A greybox fuzzer utilizes a mutator to automatically generate inputs to test the program. Unlike a random input generator, greybox fuzzer also monitors the program behavior to determine if the generated input triggers a new behavior. Inputs that trigger new behaviors are saved for future mutation. This monitoring is simple yet effective in practice. As a result, much work have focused on different parts of the fuzzer to improve its overall performance and applications.

Despite its popularity, some aspects of greybox fuzzing and its applications have not been thoroughly studied. In this thesis, we cover three aspects of greybox fuzzing. First, many fuzzers aim to increase branch coverage. However, high branch coverage is only a sufficient condition for triggering bugs. We revisit some designs of the fuzzing process to increase the likelihood of finding bugs. We first design a tool called Integrity. Integrity sanitizes integer operations within the program, which are harder to spot compared with memory errors. Integrity has discovered eight new integer errors in open-source programs. While randomized fuzzers excel at increasing branch coverage, they struggle with solving predicates set by Integrity. To trigger bugs more effectively, we propose a deterministic fuzzer Valkyrie. Valkyrie uses principled approaches, such as gradient descent and compressed branch coverage, to eliminate the randomness in fuzzers while increasing throughput. Our evaluation shows that Valkyrie can find bugs faster than the state-of-the-art in many cases.

Second, generic fuzzing is often less effective than specialized fuzzing. By incorporating expert knowledge into the fuzzer, a specialized fuzzer can reach deeply nested code more quickly. We select the LLVM backend as a test bed to see if a specialized strategy can find bugs in compilers. We develop IRFuzzer with a tailored mutation and monitoring method customized for the LLVM

backend. We model LLVM intermediate representation (IR) so that IRFuzzer guarantees to generate valid input for the LLVM backend. IRFuzzer monitors matcher table coverage to track "behavior" in a more fine-grained manner with little overhead. IRFuzzer has found 78 new bugs in upstream LLVM, with 57 of them fixed, five of which have been backport to LLVM 15. These findings demonstrate that specialized fuzzing provides useful, actionable insights to LLVM developers.

Finally, fuzzers generate large quantities of inputs as a byproduct, which are often discarded after the fuzzing process is completed. These inputs trigger different behaviors of the program. We notice that these behaviors can be vital for training large language models (LLMs). With this observation, we propose using source code coupled with their test cases for LLM training, where each test case is composed of a fuzzer-generated input and its corresponding output. We first build a dataset on top of an existing one by pairing test cases. Then, we develop methods to fine-tune a trained model and pretrain a new model on this dataset. With this new training scheme, we contribute a new code understanding model, FuzzPretrain. Our evaluation shows that FuzzPretrain yielded more than 6%/19% mean average precision (mAP) improvements on code search over its baseline trained with only source code or abstract syntax trees (AST), respectively.

Acknowledgments

During the reception at the 2023 LLVM Developers' Meeting, I was talking to Steven, my manager at AMD, and his friend Max. I was asking how to become Chris Lattner, as I felt that I couldn't contribute as much as he did in his master thesis. That eventually lead to a discussion of what it really means get a PhD degree. Steven, being another PhD who is twice my age, said something that struck me:

A PhD is all about break it down and build it up again.

Looking back, many people contributed to the "break down" and "build up" process in various ways. They might not even realize it, but their help turned out to be invaluable to me. As such, I am listing some of these individuals here as a formal thank you.

I was working on Integrity when Professor Hao Chen offered me a PhD position. I agreed, thinking that doing research is pretty fun (foolish me). This position turned out to be more dedicated than I expected, as Hao spent many sleepless nights with me reviewing my papers. I have learned a lot from him: from research ideas to project management, those are all lessons that I couldn't learn elsewhere. Many of his insights also contributed to this thesis.

I have to thank my lifelong friend Yaguxun Gao: achieving this degree would have been impossible without her. We spent countless hours talking about life and feelings, playing mobile games, and walking her dog Neko, when I was home. Even though she is on the other side of the planet, her words always find a way to build me up over and over again during my darkest times. She encouraged me and trusted me like no one ever has, and for that, I will always be grateful.

I was struggling on research when Steven Neuendorffer offered me an internship at AMD. It was a fun and productive summer that changed the course of my PhD. I learned more about compilers, contributed to the open-source community, and connected with more compiler experts that I wouldn't have known otherwise. Our work together eventually led to IRFuzzer, which gave me a boost of confidence that I could finish this. Moreover, this internship also greatly aided my job hunting: it wouldn't have be so successful without the work we did.

Professor Fu Song sparked my interest in compiler technologies. He taught *Compilers* and *Theory of Computation*, which encouraged me to set foot in the area of compilers and got interested in LLVLM during my undergraduate years. He also provided valuable advice during my PhD studies.

Many hardships broke me down during my PhD study. I survived them so I could build myself up again. Those are the life lessons that I had to learn the hard way. As I like to phrase it, they were not *obstacles*, they were *training data*, and now the model is **trained**.

Chapter 1

Introduction

1.1 Greybox fuzzing

Reliable software is vital to our society. Despite the efforts spent to enhance the robustness and reliability of software, such as unit testing, model checking, and static analysis, vulnerabilities still remain in many codebase. However, these approaches do not scale very well as they require expert knowledge of how the software works. To solve this issue, an automated approach is necessary.

In 2014, American Fuzzy Lop (AFL)[1] was proposed. Unlike random input generators, which are not effective in program testing, AFL introduced a "behavior monitor," or coverage monitor, to determine if a new input triggered new program behaviors. If so, the input is saved for future mutation. Because AFL only uses a modified compiler without examining the source code, this approach is often referred to as *greybox fuzzing*.

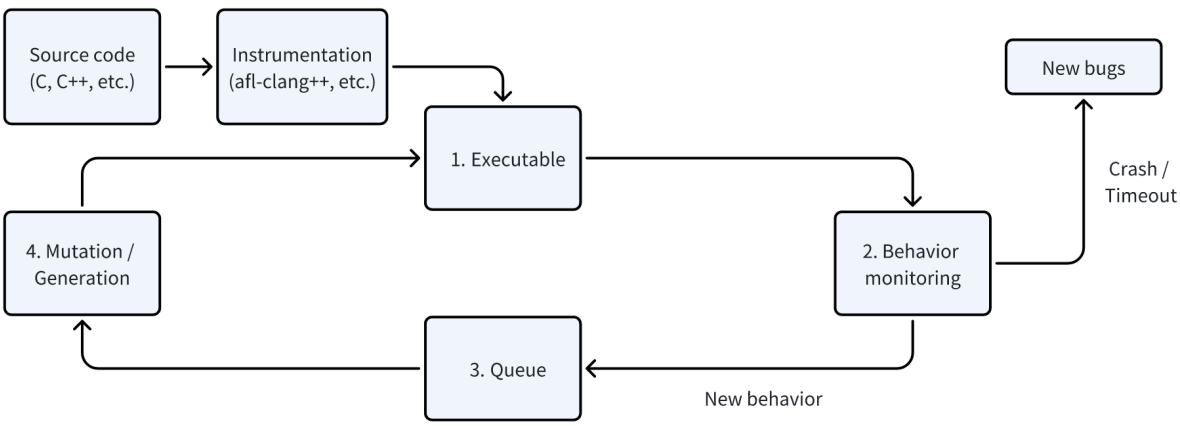


Figure 1.1: The workflow of greybox fuzzers.

Formally, greybox fuzzing consists of four stages, as shown in Figure 1.1. First, a program is compiled with a modified compiler that *instruments* the source code to automatically track code coverage. AFL uses `afl-clang` or `afl-clang++` to assign a hash ID to each edge in the control flow graph (CFG). Whenever an edge is executed, the corresponding counter will increment to reflect that behavior. If the program crashes unexpectedly or times out, the input will be saved for further investigation, as these often represent bugs in the program. Otherwise, if the program exits normally, a coverage report is generated. The behavior monitor uses these reports to determine if a new behavior has been observed. Different fuzzers can define "behavior" differently: AFL considers only branch coverage when determining if there is a new behavior. As we will see in Chapter 4, other metrics are used to define new behaviors. When a new behavior is triggered, the new input (or *seed*) is stored in a queue for future testing. AFL uses a simple first-in-first-out queue to store seeds. However, in Chapter 2 and Chapter 3, we will see that it is necessary to prioritize some seeds over others. The mutation strategy takes stored seeds to apply further random mutation. Various methods have been developed and deployed, including bit flipping, byte incrementing, slicing, etc.

Despite its simplicity, after some time of fuzzing, AFL can generate seeds that are valid and can be effectively applied to different applications. Fuzzers can be used to detect bugs in compilers [2–5], smart contracts [6, 7], or Web APIs [8, 9]. Researchers have also found ways to fuzz libraries with

a simple setup [10–13]. Operating systems can be fuzzing with proper settings [14–18], as well as IoT devices [19–21]. Even hardware can be modeled as software so it can be fuzzed [22]. Machine learning models have been researched alongside fuzzers: they are either subject to fuzzing [23–26], or have emerged as part of the fuzzer [27–29].

Since then, many researchers have conducted studies to improve the effectiveness of AFL, including all stages of AFL's execution. Some work has investigated how to use less instrumentation to reduce the execution overhead [30–34]. Researchers have also studied more effective behavior monitoring approaches other than branch coverage [35, 36]. Different methods have been proposed to prioritize seeds to improve the performance of fuzzing [37–42]. More advanced mutation strategies, especially grammar-based strategies, have been developed and demonstrated to enhance fuzzing performance compared with random mutation [43–46]. Many aspects of fuzzing and its applications are still neglected. In the following sections, we will cover three aspects of greybox fuzzing and its applications that have not been well studied.

1.2 Improving fuzzing performance using principled techniques

Covering more branches is only a necessary condition for triggering bugs. Fuzzers need a signal better than branch coverage to know that a bug has been reached, and thus spend more energy to trigger it. Sanitizers [47] serve as a good oracle for bugs, and some work rely on sanitizers [41, 48] to guide fuzzing. However, despite the existence of undefined behavior sanitizers [49, 50], integer errors are often not targeted in many fuzzing practices.

Integer arithmetic errors are a major source of software vulnerabilities. Since they rarely cause crashes, they are unlikely to be found by fuzzers without special techniques to trigger them. In Chapter 2, we design and implement Integrity, which finds integer errors using fuzzing. Our key contribution is that, through targeted instrumentation, we empower fuzzers with the ability to trigger integer errors. In our evaluation, Integrity found all the integer errors in the Juliet test suite [51] with no false positive. On nine popular open-source programs, Integrity discovered a

total of 174 true errors, including eight crashes and 166 non-crashing errors. A major challenge during error review is to efficiently determine if a non-crashing error was harmful. While solving this problem precisely is challenging because it depends on the semantics of the program, we propose two methods to identify potentially harmful errors, based on the statistics of traces produced by the fuzzer and by comparing the output of independent implementations of the same algorithm. Our evaluation demonstrates that Integrity is effective in finding integer errors.

On the other hand, we find that Integrity often cannot solve some integer constraints we instrumented. After investigation, we believe that instrumentation and mutation strategies call for a revisit for two reasons. First, branch coverage feedback that is based on random edge ID can lead to branch collision, i.e., two edges share the same ID, lowering the effectiveness of behavior monitoring. Besides, state-of-the-art fuzzers heavily rely on randomized methods to reach new coverage. These approaches may be useful to improve branch coverage, but cannot effectively solve constraints instrumented by Integrity. Even Angora [44], which advocates for a mutator based on gradient descent, often turns to heuristics and randomness as a last resort.

We believe deterministic techniques deliver consistent and reproducible results. To validate our belief, we propose Valkyrie in Chapter 3. Valkyrie is a greybox fuzzer whose performance is primarily boosted by deterministic techniques. Valkyrie combines collision-free branch coverage with context sensitivity to maintain accuracy while introducing an instrumentation removal algorithm to reduce overhead. It also pioneers a new mutation method, the compensated step, allowing fuzzers that use solvers to adapt to real-world fuzzing scenarios without randomness. Expanding Integrity's idea, Valkyrie proactively identifies possible exploit points in target programs, including not only integer errors but also memory errors. Valkyrie utilizes its solver to trigger the possible exploits. We implement and evaluate Valkyrie on the standard benchmark Magma [52], and a wide variety of real-world programs. Valkyrie triggered 21 unique integer and memory errors, 10.5% and 50% more than AFL++ [53] and Angora [44], respectively. Valkyrie reached 8.2% and 12.4% more branches in real-world programs, compared with AFL++ and Angora, respectively. We also verify that our branch counting and mutation method is better than the state-of-the-art, which shows that

deterministic techniques trump random techniques in consistency, reproducibility, and performance.

1.3 Specialized fuzzing for the LLVM backend

Many strategies focus on generic methods [1, 53, 54], meaning the fuzzer does not have any prior knowledge about the program under test. However, it has been shown in multiple applications that a tailored fuzzer tends to find more vulnerabilities [55–58]. As such, our aim in Chapter 4 is to fuzz the backend of the LLVM [59]. Due to the complexity of LLVM, manual testing is unlikely to suffice, yet formal verification is difficult to scale. End-to-end fuzzing can be used, but it has difficulties in achieving high coverage of some components of LLVM.

In Chapter 4, we implement IRFuzzer to investigate the effectiveness of specialized fuzzing of the LLVM compiler backend. We focus on two approaches to improve the fuzzer: guaranteed input validity using constrained mutations and improved feedback quality. We model LLVM IR using a descriptive language so that the mutator in IRFuzzer can generate a wide range of LLVM IR inputs, including structured control flow, vector types, and function definitions. IRFuzzer also instruments coding patterns in the compiler to monitor the execution status of instruction selection. The instrumentation provides a new coverage feedback called matcher table coverage, which can be used to determine if a new behavior is triggered. It can also provide an architecture specific guidance to the mutator. As a result, even if IRFuzzer is architecture agnostic, it is capable of fuzzing *all* backend targets in LLVM.

We show that IRFuzzer is more effective than existing fuzzers by fuzzing. We conduct evaluation on 29 mature LLVM backend targets. Comparison with end-to-end fuzzers like CSmith [60] and GrayC [3] demonstrates the necessity of specialized fuzzing. Our evaluation also demonstrates how valid input mutation and matcher table coverage individually contribute to the end results. In the process, we reported 78 confirmed new bugs in LLVM upstream, out of which 57 have been fixed, and five have been backport to LLVM 15. These findings shows that specialized fuzzing provides useful, actionable insights to LLVM developers.

1.4 Enhancing program understanding using fuzzer generated test cases

While the halting problem is undecidable [61], the semantic understanding of programs has attracted significant attention in the community, as it can be applied to traditional software engineering tasks like testing, bug analysis, code repair [62–64]. Inspired by the recent successes of large language models (LLMs) in natural language understanding, tremendous progress has been made by treating programming languages as another form of natural language and training LLMs on program code [65–67]. However, programs are fundamentally different from texts, as they are structured and syntax-strict to be properly compiled or interpreted to perform a desired set of behaviors given any inputs. Existing work benefits from syntactic representations to learn from code less ambiguously in the forms of abstract syntax tree (AST), CFG, etc. However, programs with the same purpose can be implemented in various ways, showing different syntactic representations, while those with similar implementations can have distinct behaviors. It is hard, if not impossible, to capture all program behaviors in one dataset. On the other hand, fuzzers generate a large amount of inputs as a byproduct, which are often discarded after fuzzing is completed. We hypothesize that these inputs generated by the fuzzers can be useful to LLMs as they can trigger different program behaviors. Therefore, we propose to incorporate the relationship between the program, its inputs, and corresponding outputs into learning, in order to achieve a deeper semantic understanding of programs.

In Chapter 5, we first construct two new datasets based on POJ104 [67] and CodeNet [68]. We apply AFL++ on the programs contained in these two datasets and couple the source code with the inputs generated by the fuzzer. Then we use these data to fine-tune a pre-trained LLM. We verify the effectiveness of the proposed method on two program understanding tasks, including code clone detection and code classification, and it outperforms the current state-of-the-art by large margins.

As a follow-up, we wish to pretrain a new model to incorporate this semantic information better. Hence, in Chapter 6, we propose FuzzPretrain to explore the dynamic information of programs

revealed by their test cases and embed it into the feature representations of code as complements. We carefully design a distillation process so that the test case is only required during pre-training. FuzzPretrain yielded more than 6%/19% mean average precision (mAP) improvements on code search over its counterparts trained with only source code or AST, respectively. Our experiments demonstrate the benefits of learning discriminative code representations from program executions.

Chapter 2

Integrity: finding integer errors by targeted fuzzing

2.1 Introduction

Integer arithmetic errors are a significant source of security vulnerabilities [69]. Integer overflow and underflow[1] are undefined behavior in many languages, such as C/C++, and may cause security check bypass or malicious code execution. For example, on April 22, 2018, attackers created a massive number of Beauty Coins (BEC) in two transactions by exploiting an integer overflow in ERC20 [71], which forced the exchange platform OKEx to roll back all the transactions two days later [72]. Divide-by-zero causes the program to crash and so may be used to launch denial of service attacks. The number of reported integer arithmetic bugs has been increasing rapidly in recent years, which account for 104, 232, and 635 Common Vulnerabilities and Exposures (CVE) in 2016, 2017, and 2018, respectively.

Prior work showed how to detect integer overflows *when* they happen. For example, Integer Overflow Checker (IOC)[49, 50], which has been included in Undefined Behavior Sanitizer (UB-

[1]The term *underflow* sometimes refers to float point underflow. However, in accordance with Common Weakness Enumeration (CWE) [70], in this chapter "underflow" means that the result of an integer arithmetic operation is smaller than the smallest value that the type can represent.

Table 2.1: Verified, unique arithmetic errors that Integrity found in real world applications, compared with Angora + UBSan. Note that the total numbers of unique errors at the bottom are fewer than the sums of the rows above because some programs share the same library and therefore we removed these duplicate errors when calculating the totals.

Program	Errors found by Integrity			Errors found by	Improvement
	Crashing	Non-crashing	Total(I)	Angora + UBSan(A)	(I - A)
cjpeg	1	12	13	0	+13
djpeg		17	17	14	+3
file		17	17	0	+17
img2txt	3	21	24	2	+22
jhead	2	4	6	4	+2
objdump		5	5	0	+5
readelf		38	38	0	+38
tiff2ps		27	27	1	+26
tiffcp	2	31	33	2	+31
Total	8	166	174	23	+151

San) [73] since LLVM 3.5. However, they relied on the programmer to manually create test cases to trigger those bugs, which is laborious and unreliable. We face the challenge of how to generate these test cases automatically and efficiently.

Fuzzing is an automated approach for finding software bugs. Starting with AFL, graybox fuzzers have made great strides in finding bugs fast. They instrument programs with the code for recording program state during execution and use that information to guide input mutation. Fuzzers differ in their strategies for *exploration*, which aims at expanding branch coverage. Previous exploration strategies include matching magic bytes [74], finding sanity checks and checksums [75, 76], measuring the distance between the input and target location [77, 78], and solving constraints like Angora [44]. Besides exploration, another goal of fuzzing is *exploitation*. In the context of fuzzing, exploitation refers to triggering bugs, regardless if the bug may be used to launch attacks. It is difficult to find good exploitation strategies. As a result, most fuzzers randomly mutate the input to hope that some mutated input might trigger bugs. Given the huge space of input, the probability that a randomly mutated input will trigger a bug is low. Moreover, fuzzers have difficulty in detecting bugs that do not crash the program because they lack reliable signals that indicate bugs in those cases. For

example, arithmetic errors cause a program to misbehave (e.g., to produce wrong results), but they rarely cause the program to crash.

Our goal is to allow fuzzers to exploit integer arithmetic errors efficiently. Our key technique is to provide fuzzers with critical information by targeted instrumentation such that the information can later be used to guide fuzzers to exploit potential bugs. For example, to detect overflow when adding two 32-bit signed integers, we extend both the operands to 64 bits, compute their sum (which cannot overflow), and, if the sum is out of the range of 32-bit signed integers, execute a special *guard branch* to send a signal to the fuzzer to indicate the error. This way, if the fuzzer can reach the guard branch, then an integer overflow occurs. The same idea can be used to check for other bugs, such as index out of range, null pointer dereference, etc.

In principle, the above idea works with any fuzzer. However, to find bugs efficiently, we need to overcome three challenges. First, we need to select a fuzzer that efficiently solves the constraints indicating arithmetic errors (Section 2.2.2.1). Second, the guard branches inserted by the fuzzer have much lower expected reachability than the original branches, because the guard branches indicate arithmetic errors but most arithmetic operations should not have such errors. Therefore, we need to redesign the fuzzer's scheduling algorithm to assign different priorities to the original and guard branches, respectively (Section 2.2.2.2). Finally, we need to send a unique signal to the fuzzer to indicate arithmetic errors if the guard branches are explored. The fuzzer should let the program continue exploring branches after receiving the signal, in contrast to when the signal indicates a memory violation (Section 2.2.2.3).

It might be tempting to implement the above idea by simply combining a sanitizer (e.g., UBSan [73]) with a fuzzer. However, because of the challenges described above, such a naive combination would result in poor performance, as we will show in Section 2.4.4. Instead, we implemented our approach in a tool called Integrity. As we will show in Section 2.4, Integrity is effective in finding integer arithmetic errors in both standard test suites and popular open-source programs. On the Juliet Test Suite [51], Integrity found all the bugs with no false positive (Table 2.2). Table 2.1 shows the bugs that Integrity found on 9 popular open-source programs from 6 packages.

In total, Integrity found 174 unique arithmetic errors, where 8 caused crash but 166 did not. We define a unique error by a unique (file name, line number, column number) tuple.

Fuzzing is attractive because it provides inputs that witness errors. When an error caused a crash, there is no doubt that the program misbehaved. However, when the error did not cause a crash, verifying whether the error caused the program to misbehave becomes difficult as the decision must take domain knowledge into consideration. We made progress on this problem by proposing two methods. The first method is based on the statistics of the traces generated by the fuzzer. If an integer arithmetic error occurred on most traces generated by the fuzzer where the arithmetic operation executed, then the error was likely benign, as long as the fuzzer had adequate path coverage. The other method is based on comparing the output of independent implementations of the same algorithm on the same input. If an integer error caused one implementation to misbehave, then the other independent implementation of the same algorithm will unlikely generate a similar output, as long as the output is a deterministic function of the input. These two approaches, when applicable, call attention to integer errors that are potentially harmful.

2.2 Design

Fuzzers mutate the input to find bugs in the program. They have two goals: (1) exploration: explore different paths; and (2) exploitation: trigger bugs (regardless whether they can be used to launch attacks). Previously, fuzzers were used predominantly to find memory errors. To use fuzzers to find integer arithmetic errors effectively, we need to modify both their exploration and exploitation strategies.

2.2.1 Exploitation

2.2.1.1 Arithmetic operations

We detect integer overflow and underflow during addition (`+`), subtraction (`-`), multiplication (`*`), shift left (`<<`), and divide by zero during division (`/`) and remainder (`%`). We instrument LLVM

IR code to detect those errors as follows.

- `+`, `-`, `*`: We promote both the operands to the next longer type (e.g., from `int32_t` to `int64_t`, and from `uint32_t` to `uint64_t`), evaluate the expression in the longer type, and check if the result is out of the range of the original type. As long as the width of the next longer type is as least doubled (e.g., `int8_t`, `int16_t`, `int32_t`, `int64_t`), which is the case in C and most C-like languages, the operation in the longer type never overflows. For example, to check if `(int8_t)x + (int8_t)y` overflows, we compute `(int16_t)x + (int16_t)y` and check if the sum is out of the range of `int8_t`.

- `<<`: A left shift operation `x << n` overflows if and only if $\text{hp}(x) + n$ is greater than or equal to the width of (number of bits in) the result type, where the function $\text{hp}(x)$ is the position of the highest non-zero bit of x. For example, $\text{hp}(0b00000001) = 0$, $\text{hp}(0b10000000) = 7$.

- `/` and `%`: We check if the second operand is 0. For `/`, we also check if the operands are MININT and -1 because MININT / -1 = MAXINT + 1 overflows.

2.2.1.2 Range inference

Integer types have different ranges. To infer the correct integer type, we must determine both the bit width and sign.

Bit width inference For each operation, LLVM promotes every operand shorter than 32 bits to 32 bits, executes the operation, and then truncates the result back to the destination type when necessary. Therefore, if a truncation follows the operation, then we use the destination type of the truncation to infer the bit width; otherwise, we use the left-hand side of the operation.

Sign inference LLVM IR does not distinguish between signed and unsigned variables. LLVM determines if an operation on 32 or more bits may have signed overflow or unsigned overflow using the sign information from abstract syntax tree (AST), and encodes that information as a tag in the arithmetic instructions. For example, `add nsw` (no signed wrap) and `add nuw` (no unsigned wrap). We use these tags to infer the sign. However, operations on integers shorter than 32 bits carry no

such tag because they never overflow in the range of 32-bit integers. In those cases, we infer the sign of each operand using the cast operation before the arithmetic operation. When LLVM casts the shorter type to 32 bits, we examine if the cast is signed or unsigned. If both operands are cast, we take the sign of the operand of the longer type if the operands have different bit widths. If they have the same bit width, and if either operand undergoes an unsigned cast, we infer the sign of the destination type as unsigned; otherwise, we infer the sign as signed.

2.2.1.3 Instrumentation reduction

When we instrument an integer arithmetic operation to check for arithmetic errors, we create new branches. When a program has many integer arithmetic operations, the instrumentation would create many new branches for the fuzzer to explore. However, these branches differ from the original branches in the program in a very important way for the fuzzer: we expect most original branches to be reachable but few instrumented branches to be reachable (because the latter represent arithmetic errors). Since unreachable branches waste the fuzzer's computing budget, during instrumentation we eliminate branches that are guaranteed unreachable as follows:

- While we need to check both overflow and underflow of signed operations, we need not check underflow of unsigned operations, because once promoted to a wider type, underflow becomes overflow. For example, when the original type is 8-bit unsigned int, `(uint8_t)0 - 1 = 0xff` causes underflow. However, when promoted to 16-bit unsigned int, `(uint16_t)0 - 1 = 0xffff` causes an overflow on the original type because the result 0xffff is larger than the upper limit of the original type, 0xff.
- We do not check shift operation on negative integers for the same reason as above.
- When an operation is square, we do not check for underflow because it cannot.
- When a value is added to a negative constant or is subtracted by a positive constant, we do not check for overflow; similarly, when a value is added to a positive constant or is subtracted by a negative constant, we do not check for underflow.

Section 2.4.5 will show that the above optimization significantly reduced the number of branches that the instrumentation added to the program, and hence the number of constraints that the fuzzer tries to solve.

2.2.2 Exploration

The instrumentation described in Section 2.2.1 reduces the problem of exploitation to the problem of exploration. At each operation with potential integer arithmetic errors, Integrity inserts a conditional statement to check for integer arithmetic errors. When an error happens, the conditional statement executes a branch, called the *guard branch*. In principle, we can use any fuzzer to do the exploration. However, we desire to select a fuzzer that can explore arithmetic errors efficiently. Moreover, since the guard branches are inherently different from the branches in the original program (original branches), the fuzzer must treat them differently: the fuzzer should triage between the original and guard branches when scheduling branches (Section 2.2.2.2), and should behave differently between when arithmetic errors occur and when other errors occur (Section 2.2.2.3).

2.2.2.1 Fuzzer choice

Section 2.2.1 provides critical information to the fuzzer by instrumenting the guard branches that represent those errors. While we may use any fuzzer to take advantage of that information, we selected Angora [44] for its two beneficial properties.

First, Angora fuzzes individual branches and can prioritize different branches. With enough computing budget, Angora fuzzes every branch on a path at least once. Since we associate every potential arithmetic error with a guard branch, Angora exploits (tries to trigger) every arithmetic error on the path. Angora also allows us to triage different branches, which is handy because the original branches and guard branches have different expected reachability (Section 2.2.2.2).

Second, Angora's input mutation strategy fits our goal well. When fuzzing a branch, Angora uses byte-level taint tracking to identify the input byte offsets that flow into the predicate that

Algorithm 1 Integrity's scheduling algorithm.

```
function POP                                          ▷ Returns the next branch to fuzz
    return priorityQueue.pop()
end function
function PUSH(b)                                      ▷ Pushes a new or existing branch onto the queue
    if b is a newly found branch then
        if b.tag = Tag.Original then
            b.priority ← MAX_PRIORITY
        else
            b.priority ← GUARD_INIT_PRIORITY
        end if
    else
        b.priority ← b.priority − 1
    end if
    priorityQueue.push(b)
end function
```

guards the branch. Then, Angora considers the predicate as a blackbox function on those byte offsets and uses gradient descent to find an input that satisfies the predicate. When the blackbox function is linear or monotonic, this mutation strategy guarantees to find a solution quickly. `+` and `-` are linear functions, and `*` is a monotonic function. When their operands take their values directly from in the input, Angora can solve the predicates of those operations efficiently.

2.2.2.2 Branch triage

As discussed in Section 2.2.1.3, original branches and guard branches have different expected reachability: we expect most original branches to be reachable but few guard branches to be reachable because few arithmetic operations have errors. Moreover, before the fuzzer can reach an original branch b, it cannot explore any guard branch that b dominates.[2] Therefore, we must let the fuzzer assign higher priority to the original branches than to the guard branches.

We replaced Angora's scheduling with the following algorithm:

- At compile time, instrument each branch with a tag to indicate whether it is an original branch or a guard branch.

[2] A node d dominates a node n if every path from the entry node to n must go through d.

- At run time, store all the branches to be fuzzed in a priority queue.
- When finding a new branch, assign the branch a priority according to the branch tag (original or guard branch), and then push the branch onto the priority queue (`PUSH` in Algorithm 1).
- When failing to solve a branch, lower the priority of the branch and push it onto the priority queue (`PUSH` in Algorithm 1).
- When ready to explore a new branch, call `POP` in Algorithm 1 to get the branch with the highest priority.

2.2.2.3 Signal of errors

When the fuzzer receives a signal indicating an error in the program, it stops the program execution and records the input, and the error and its location. Memory access violation, such as segmentation fault, is the most common signal. To reuse this framework, Integrity lets the instrumented branches send a pre-determined signal to the fuzzer to indicate arithmetic errors.

However, merely sending a signal would be inadequate. Fuzzers stop the program when receiving signals. It makes sense when the signal is triggered by a memory error because the program cannot continue anyway. However, when the signal is triggered by an arithmetic error, the fuzzer should let the program continue to explore more paths, particularly when the error is false positive (see Section 2.4.2.1 for examples). Without this ability, a false positive arithmetic error early in the program would prevent the fuzzer from exploring most paths because most paths descend from the location of that error. We implemented this desirable function in Angora.

2.3 Implementation

We implemented Integrity as an LLVM pass in 924 lines of C++. We also modified Angora to do branch triage (Section 2.2.2.2) and to deal with the new signal of arithmetic errors (Section 2.2.2.3) in 3419 lines of Rust.

We found that some programs may use 64-bit types (`uint64_t`, for example). However, Angora supported only 64-bit constraints, which was inadequate to check the overflow of the arithmetic operation on two 64-bit integers. To tackle this problem, we extended Angora to support 128-bit constraints. We did so by using `u128` and `__uint128_t` in Rust and C, respectively. In the case of a 128-bit or higher precision integer operation, we created a new struct that has two (or more) 128-bit unsigned integers inside and implemented all the arithmetic traits (`Add`, `Sub`, `Mul`, etc.) for it.

Regardless, in this chapter we deal with `general` arithmetic operation errors instead of big integer errors, which should've done by using our tool to fuzz bit integer libraries like GNU multiple precision arithmetic library [79].

2.4 Evaluation

We evaluated the performance of Integrity on both the Juliet test suite [51] and popular open-source programs. We also evaluated the impact of instrumentation reduction described in Section 2.2.1.3.

All our experiments ran on a Linux server with two Intel Xeon Gold 5118 CPUs and 256 GB RAM.

We set $MAX\_PRIORITY$ and $GUARD\_INIT\_PRIORITY$ in Algorithm 1 to 65,535 and 65,534, respectively, to guarantee that the fuzzer will try to solve all the original branches at least once before solving the guard branches.

2.4.1 Juliet test suite

The Juliet test suite, developed by the National Security Agency (NSA), contains tests for errors listed in Common Weakness Enumeration (CWE) [70]. It organizes the tests in a hierarchy: at the top level, the suite contains one test set for each CWE. Then, each test set contains many subsets, and each subset contains many tests. Each test is a C or C++ program containing a carefully designed and inserted error. This test suite provides ground truth for evaluating the false positive

and false negative of Integrity.

We used Juliet Test Suite v1.3 and selected the following test sets relevant to integer arithmetic errors:

- `CWE190_Integer_Overflow`
- `CWE191_Integer_Underflow`
- `CWE194_Unexpected_Sign_Extension`
- `CWE197_Numeric_Truncation_Error`
- `CWE369_Divide_by_Zero`

We excluded the following tests in the above test sets:

- Deterministic errors: These errors always happen regardless of the input, e.g., overflow caused by constant integers.
- Floating point errors, since we focus on integer arithmetic errors only.
- C++ programs. As discussed in Section 2.2.2.1, we used Angora as the fuzzer, and currently it supports only C programs. This is not an inherent limitation of Integrity.

Two CWEs related to integer arithmetic errors are worth mentioning. One of them is `CWE197_Numeric_Truncation_Error`. Integer truncation causes an error when the result is out of the range of the destination type. Therefore, to detect this error accurately, we must detect the destination type (both sign and width) accurately. For example, consider `x & 0x0000ffff`. If the destination type has more than 16 bits or if it is unsigned 16-bit integer, then no overflow can happen. In all the tests of `CWE197`, it is easy to infer the destination types accurately because of the way how those errors were injected. However, in real world programs, we found that accurately inferring the destination type in the context of integer truncation was difficult. Therefore, we disabled this rule when checking real world programs in Section 2.4.2.

Table 2.2: Errors that Integrity found on the Juliet test suite. A "-" cell means that the corresponding test set on the top contains no corresponding subset on the left. Integrity found all the errors with no false positive. Every test contains one inserted arithmetic error except subset s02 of CWE197, where half of its inserted bugs contain two truncation errors each.

	CWE190		CWE191		CWE194		CWE197		CWE369	
Subset	bugs added	bugs found	bugs added	bugs found	bugs added	bugs found	bugs added	bugs found	bugs added	bugs found
s01	114	114	76	76	304	304	152	152	112	112
s02	38	38	38	38	0	0	76	**114**	38	38
s03	190	190	114	114	-	-	-	-	-	-
s04	114	114	190	190	-	-	-	-	-	-
s05	114	114	190	190	-	-	-	-	-	-
s06	190	190	-	-	-	-	-	-	-	-
s07	190	190	-	-	-	-	-	-	-	-

```
1  short CWE_197_s02_trunc_twice(char* inputBuffer){
2    short data = 0;
3    if (fgets(inputBuffer, 14, stdin) != NULL) {
4      data = (short)atoi(inputBuffer);
5    }
6    return (char) data;
7  }
```

Listing 2.1: A test in CWE197 s02, which contains two truncation errors on Line 4 and Line 6.

The other one is `CWE680_Integer_Overflow_to_Buffer_Overflow`. This error happens when calling the function `malloc(site_t)` and when `size_t` is defined by `uint32_t`, which occurs on only 32-bit platforms. Since the fuzzer that we used(Angora) ran only on 64-bit platforms, we did not test this error.

Table 2.2 shows that Integrity found all the bugs in the test sets of the above five CWEs with no false positives. Every test case has one inserted arithmetic error except subset s02 of CWE197. This subset contains 76 tests, where half of the tests contains two truncation errors each as shown in Listing 2.1: first truncating the result of `atoi` into `short`, and then further into `char`, both of which cause truncation errors. Therefore, Integrity found a total of $38 + 38 \times 2 = 114$ unique errors in this subset of 76 tests.

We tried Angora and Angora + UBSan on this test set, respectively. Neither of them found any

bugs.

2.4.2 Real world applications

We evaluated Integrity on popular real world applications. We selected 9 applications from 6 packages that have many integer operations, such as image processing and executable file parsing. Detailed version and command line arguments are shown in Table 2.3. On each program, we ran Integrity on 12 cores for 72 hours.

Table 2.3 shows all the unique errors that Integrity found. We identified a unique error by the (file name, line number, column number) tuple where the error occurs. We divide those errors into three categories. The first category contains all errors that caused crashes (Section 2.4.2.2). Then, we manually reviewed the remaining errors to identify benign ones. We determined an error to be benign when we found ~~strong evidence~~ that the error did not cause the program to misbehave (Section 2.4.2.1). After excluding those benign errors, the remaining errors belong to the non-crashing error category (Section 2.4.3).

It is also worth mentioning that *tiff2ps* and *tiffcp* share the same underlying library(*libtiff*). As a result, Integrity found 6 duplicate non-crashing errors and 19 benign errors in both program. We removed those duplicate errors from total error count in Table 2.1 and Table 2.3.

2.4.2.1 Benign errors

An error is benign when we found strong evidence that the error had been expected by the programmer and therefore did not cause the program to misbehave. We classify all the benign errors found into two classes:

Intentional overflows The programmer intended to use the result of an overflown value. One example is `v << (32 - b) >> (32 - b)`, where the programmer intended to exact the lower b bits from the unsigned 32-bit integer v, and implemented it by shifting v by $32 - b$ bits to the left and then shifting by $32 - b$ bits to the right. As long as b is in (0, 32], the implementation correctly achieved the programmer's goal, even though overflow might happen during the left shift.

Table 2.3: Unique errors that Integrity found in common open-source programs. Note that the total numbers of unique errors at the bottom are fewer than the sums of the rows above because when calculating the totals we removed the duplicate errors in the libraries shared by different programs.

Package	Version	Program	Unique errors			
			Divide by zero	Overflow to crashing	Non-crashing	Benign
libjpeg-ijg	v9a	cjpeg	1		12	63
		djpeg			17	101
file	5.32	file			17	7
libcaca	0.99beta99	img2txt	1	2	21	36
jhead	3.00	jhead		2	4	4
binutils	2.29	objdump -x			5	11
		readelf -a			38	27
libtiff	4.0.7	tiff2ps			27	36
		tiffcp -i	2		31	49
Total			4	4	166	315

Unused overflown values This class of benign errors is commonly introduced by compiler optimization.

```
while (i--) { /* loop body */ }
```

is an example, Listing 2.2 shows the compiled LLVM IR. The loop subtracts 1 from the loop variable (an unsigned integer) and saves the result in another variable just before checking the predicate that if the loop variable is not 0. When the loop variable is 0, the subtraction underflows, but its result will never be used because the loop finishes.

2.4.2.2 Crashes

Arithmetic errors may cause crashes in two different ways. Divide by zero causes a crash immediately, while overflown or underflown values may cause a crash when used as indices to arrays. Integrity discovered eight crashes, among which four are divide by zero, and four are overflow.

Listing 2.3 shows a divide by zero error on Line 4 in the program *libjpeg-ijp*. Integrity found an input that caused the parameter `samplesperrow` to become 0, which then caused divide by zero on

```
1  ; <label>:loop_head:
2  %loop_var = load i32, i32*%loop_ptr, align 4
3  %next_loop_var = add nsw i32 %loop_var, -1
4  store i32 %next_loop_var,i32*%loop_ptr,align 4
5  %cond = icmp ne i32 %loop_var, 0
6  br i1 %cond, label %loop_body, label %loop_end
7  ; <label>:loop_body: /* body */
8  br label %loop_head
9  ; <label>:loop_end:
```

Listing 2.2: An example of benign integer overflow. After LLVM optimization passes, the C program was translated into the IR shown in the figure, the syntax slighted modified for readability. On Line 3, the `add` instruction overflows when the loop variable `%iter_var` is 0, but the overflown result will never be used.

```
1  // jmemmgr.c:395~435
2  ... alloc_sarray(..., unsigned samplesperrow, ... ) {
3    ...
4    ltemp = ... / ((long) samplesperrow * SIZEOF(JSAMPLE));
5    ...
6  }
```

Listing 2.3: Divide by zero error in *jmemmgr.c* of *libjpeg-ijg* happens when the parameter `samplesperrow` is zero.

Line 4.

2.4.3 Which non-crashing error is harmful?

An error is said to be *harmful* when it triggers unexpected behavior, e.g. to produce a wrong result. Harmful errors may or may not be exploitable in the context of software security, yet they still cause problems in software correctness and reliability. If an arithmetic error causes a crash, it is definitely a harmful error. However, when it does not cause a crash, it is non-trivial to validate whether it is harmful.

We manually inspected all the 481 non-crashing errors reported by Integrity and determined that 315 (or 65 %) were benign. However, manual inspection is tedious and unscalable.

Automatically determining if an arithmetic error is harmful is challenging because it depends on the semantics of the application. Nevertheless, we made progress on this problem by proposing two methods, one based on statistics of the traces generated by the fuzzer, and the other based on

comparing the output of independent implementations of the same algorithm on the same input. Although they do not *prove* whether each error is harmful, they provide *evidence* that some errors are harmful and some are benign. We believe that the evidence is strong, and we plan to quantify it in a probabilistic framework in future work. These two approaches, when applicable, call attention to integer errors that are potentially harmful.

2.4.3.1 By statistics of traces

This method is based on the conjecture that a harmful bug in a popular open-source program unlikely occurs during most executions, because otherwise it would have been noticed, reported, and fixed with high probability. By this conjecture, if an integer arithmetic error occurred on most traces generated by the fuzzer where the arithmetic operation executed, then the error was likely benign, as long as the fuzzer had adequate path coverage.

To implement the above idea, for each non-crashing arithmetic error, we measured its rate of occurrence on all the traces where the arithmetic operation occurred. When this rate is above a threshold, we consider this error to be benign. We used the benign errors that we manually determined in Table 2.3 as the ground truth. Then, at each threshold, we counted the number of benign errors using the rule above, and calculated precision and recall based on the ground truth. That is, let G be the set of benign errors that we manually determined, and S be the set of benign errors that we identified by the statistics of traces. Then precision is $\frac{|S \cap G|}{|S|}$ and recall is $\frac{|S \cap G|}{|G|}$.

Table 2.4 shows the number of benign arithmetic errors and their precision and recall with regard to the ground truth. The overall precision is 79.2% at the threshold of 0.95, and is 75.7% at the threshold of 0.70. The overall recall is 37.5% at the threshold of 0.95, and is 67.3% at the threshold of 0.70. On several programs, this method was quite accurate. For example, at the threshold of 0.95, this method achieved both 100% precision and 100% recall on *jhead*, and 100% precision on *cjpeg*. On 7 out of 9 programs the precision reaches above 80%, which indicates that our method can efficiently rule out part of benign error and thus reduce human labor.

Table 2.4: Benign arithmetic errors determined by statistics of traces. We use the benign errors found by manual inspection as the ground truth when calculating the precision and recall of the benign errors determined by statistics of traces.

Program	Benign errors found by manual inspection	Benign errors determined by statistics of traces					
		Threshold=0.95			Threshold=0.70		
		Count	Precision	Recall	Count	Precision	Recall
cjpeg	63	8	100.0 %	12.7 %	48	87.5 %	66.7 %
djpeg	101	19	100.0 %	18.8 %	42	97.6 %	40.6 %
file	7	6	83.3 %	71.4 %	8	87.5 %	100.0 %
img2txt	36	18	88.9 %	44.4 %	39	59.0 %	69.9 %
jhead	4	4	100.0 %	100.0 %	5	80.0 %	100.0 %
objdump	11	12	83.3 %	90.9 %	12	83.3 %	90.9 %
readelf	27	28	71.4 %	74.1 %	36	72.2 %	96.3 %
tiff2ps	36	25	88.0 %	61.1 %	37	62.2 %	63.9 %
tiffcp	49	46	67.4 %	63.3 %	53	67.9 %	73.5 %
Total	315	149	79.2 %	37.5 %	280	75.7 %	67.3 %

2.4.3.2 By comparing independent implementations

This method uses two independent implementations P and Q of the same algorithm to evaluate whether an arithmetic error is likely harmful. If P and Q (1) agree (have identical or similar output) on all the inputs that trigger no arithmetic errors but (2) disagree (have different outputs) on the inputs that trigger arithmetic errors in P, then the errors in (2) are likely harmful. This is based on the conjecture that when an input triggers a harmful arithmetic error in P, it unlikely also triggers an arithmetic error in Q, and even if it does, the two errors unlikely cause P and Q to generate similar output. Obviously, the first property above requires the output to be a deterministic function of the input, i.e., no randomness may affect the output.

We applied the above method on the program *djpeg* in the *libjpeg-ijg* package. A JPEG encoder compresses an image by (1) dividing the image into 8×8 matrices and applying discrete cosine transform (DCT) to each matrix, (2) suppressing the high-frequency signals by element-wise dividing each matrix by a predefined matrix and rounding the result to the nearest integer, and (3) discarding all the tailing zeros. The decoder reverses the above operations, where it can infer the number of

discarded zeros based on the size of the small matrix and that of the image.

Since a JPEG decoder uses floating point arithmetic, two independent decoders may create slight different outputs on the same input. However, if the difference is large, then at least one decoder is misbehaving. We measured the difference as the average L^1 distance between two images. More precisely, let

- A and B: two images of dimension $m \times n$.
- $A_{i,j}$: a 3-channel vector representing the RGB values of the pixel at (i, j)
- $A_{i,j}^{(k)}$: the value of the kth channel. This value is in the range $[0, 255]$, and $k \in \{1, 2, 3\}$.

Definition 2.4.1. The *average* L^1 *distance* between two images A and B of identical size is:

$$D(A, B) = \frac{\sum_{c \in C(A,B)} \sum_{k \in [1,3]} \mid c^{(k)} \mid}{\mid C(A, B) \mid} \tag{2.1}$$

where

$$C(A, B) = \{A_{i,j} - B_{i,j} : i \in [1, m], j \in [1, n], A[i, j] \neq B[i, j]\}$$

To evaluate whether non-crash arithmetic errors in *libjpeg-ijg* are harmful, we selected *libjpeg-turbo* as an alternative, independent implementation. *libjpeg-turbo* has the same API as *libjpeg-ijg*; however, its decoder uses SIMD instructions to accelerate arithmetic operations while *libjpeg-ijg* does not.

We prepared two sets of JPEG images as input to the decoders:

- Normal images: We randomly picked 100 JPEG images from Android system images, LaTeX testing images, *libjpeg* testing images, and GNOME 3.28 desktop images. None of them triggered arithmetic errors on either decoder.
- Exploit images: We collected images produced by Integrity that triggered arithmetic errors

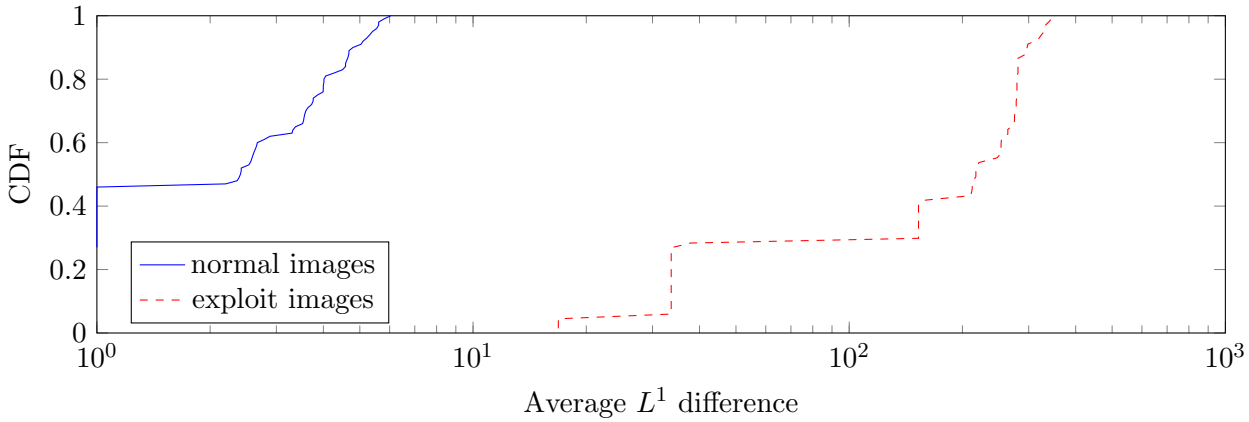


Figure 2.1: Cumulative distribution function (CDF) of the average L^1 distance (Equation 2.4.1) between the output of two decoders on the same input JPEG image. The CDF of the normal images is cleanly separable from that of the exploit images.

in the program *djpeg* in the package *libjpeg-ijg*, and then removed the following from the collection:

- Broken images: Integrity generated many images that are invalid JPEG and therefore cannot be rendered.
- Images whose width or height is less than 8 pixels. Since JPEG encoder partitions images into 8×8 matrices, the decoder's behavior on those images may be implementation-dependent.
- Images that triggered only the benign errors described in Section 2.4.2.1

After filtering, we were left with 67 exploit JPEG images.

Figure 2.1 compares the cumulative distribution functions (CDF) of the average L^1 distance (Equation 2.4.1) between normal and exploit images. The figure cleanly separates the CDF of normal and exploit images with no overlap: the L^1 distance of normal images ranges from 0.0 to 6.0 with a median of 2.4, while the distance of exploit images ranges from 16.9 to 342.4 with a median of 217.2. This implies that those arithmetic errors that Integrity found in *libjpeg-ijg* are harmful.

2.4.4 Comparison with Angora + UBSan

We compared Integrity with simple combination of Angora and UBSan. We ran Angora with UBSan in the same experimental configuration as we described in Section 2.4.2.

Table 2.1 compares the number of verified bugs found by Integrity and Angora+UBSan, respectively. Integrity found many more bugs than Angora on each program. On all program together, Integrity found 174 bugs while Angora+UBScan found only 23 bugs. Angora+UBSan found no bug in *file*, *objdump* and *readlef*, but Integrity found a total of 60 bugs in them. This result shows that Integrity performs far superior than simple combination of Angora and UBSan. Without proper information sharing (Section 2.2.2.2 and Section 2.2.2.3), the fuzzer and the sanitizer cannot cooperate well because the fuzzer would not know where the potential bugs lie and divert computation power accordingly.

As a side note, we had to overcome engineering difficulties to combine Angora and UBSan. Angora compiles two binaries for each program: one uses Data Flow Sanitizer (DFSan) [80] to do taint tracking, and the other monitors the execution traces. DFSan instruments instructions to track data flow. If the program calls a function in third-party libraries, DFSan needs a modeled function to know how to propagate the taint. When we initially compiled the programs using UBSan and DFSan, it failed because DFSan could not find the modeled functions instrumented by UBSan. [47] also warned such issues when using multiple sanitizers. We applied a temporary hack to overcome the compilation problem: we enabled DFSan and disabled UBSan when compiling the binary for taint tracking, and enabled UBSan and disabled DFSan when compiling the binary for monitoring execution traces.

2.4.5 Instrumentation reduction

To evaluate the effect of instrumentation reduction described in Section 2.2.1.3, we instrumented five libraries with and without reduction and compared the number of instrumented arithmetic operations. Table 2.5 shows that overall this technique eliminated 9% instrumented arithmetic operations.

Table 2.5: Number of instrumented arithmetic operations before and after instrumentation reduction

Library	# of instrumentation		Remaining
	after reduction	before reduction	instrumentation
libpng	2518	2773	90.80 %
binutils	16,432	18,203	90.27 %
libjpeg	14,335	15,312	93.62 %
libtiff	7383	8123	90.89 %
libpcap	714	887	80.50 %
Total	41,382	45,298	91.36 %

2.5 Related work

2.5.1 Detecting integer overflow

Integer overflow has been extensively studied [49, 50, 81–84]. IOC [49, 50] instruments AST to test for overflow. It is now part of LLVM's UBSan [73].

IOC tends to generate many benign overflows. IntEQ [81] and IntFlow [82] intend to cut down reported benign overflows. Both use the assumption that an overflown value is benign unless it is used in a sink. IntFlow combines static and dynamic analysis to determine if any overflown value flows into a sink. IntEQ relies on symbolic execution to achieve this goal. It computes a value flown into a sink in both high and low precision and compares the two values. Both these tools rely on the user to provide input (test cases) for finding overflows. Integrity overcomes this limitation by triggering arithmetic errors automatically through program instrumentation targeting arithmetic errors.

z3 [83] is a tool for solving integer-related symbolic constraints. IntScope [84] uses symbolic execution to detect integer overflow. Unlike IOC, IntScope does not rely on source code but translates x86 binary to an intermediate representation called PANDA first, then symbolically executes PANDA to detect possible arithmetic errors. Since Integrity uses fuzzing, it inherits the advantages of fuzzing over symbolic execution, such as faster execution and tolerating obscure code (e.g., external libraries, system API, etc).

2.5.2 Coverage-directed fuzzers

A coverage-directed fuzzer mutates the input to explore paths in the hope to trigger bugs on some of these paths [1, 27, 43, 44, 74, 77, 78, 85–88]. If a mutated input explores a new path, the fuzzer keeps the input as a seed. AFL [1] and LibFuzzer [86] employ evolutionary algorithms to mutate input. Driller [87] and QSYM [88] try to solve complex path constraints by concolic execution. VUzzer[74] and REDQUEEN [43] learn magic bytes and generate satisfying input without symbolic execution. bohme2017directed [77] and chen2018hawkeye [78] direct fuzzing to a set of target program locations efficiently. Angora [44] models a path constraint as a black-box function, and uses optimization methods such as gradient descent to solve it. NEUZZ [27] also uses gradient descent to explore new paths and approximates the target program's branch coverage by a neural network.

Many coverage-directed fuzzers can turn on various sanitizers to detect bugs during exploration [47, 73, 89–92]. For example, Address Sanitizer [89], Memory Sanitizer [90], Thread Sanitizer [92], and Undefined Behavior Sanitizer [73] detect invalid memory addresses, use of uninitialized memory, data races, and undefined behavior, respectively. However, those fuzzers only passively detect those bugs when they are triggered by random mutation. By contrast, Integrity instruments arithmetic operations with potential errors to triggers them actively.

2.5.3 Bug-directed fuzzers

Besides integer arithmetic errors, researchers developed fuzzers to exploit other vulnerabilities. petsios2017slowfuzz [93] targets algorithmic complexity vulnerabilities guided by resource usage. jeong2019razzer [15] guides fuzzing towards potential data races in the kernel, then deterministically triggers a race. NEZHA [94] exploits the behavioral asymmetries between multiple test programs to focus on inputs that are more likely to trigger semantic bugs. Tensorfuzz [95] use coverage-guided fuzzing methods for neural networks to find numerical errors in a trained neural network. Dowser [96] determines "interesting" array accesses that likely harbor buffer overflow, and triggers overflow by taint tracking and symbolic execution. TIFF [97] infers input types by dynamic taint analysis,

and sets input bytes with defined interesting values based on its type to maximize the likelihood of triggering memory-corruption bugs. Compared with those fuzzers, which were built to detect those specific bugs, Integrity reduces the problem of exploitation to the problem of exploration, and therefore can work with most fuzzers and can benefit from the advances of exploration technologies.

Chapter 3

Valkyrie: improving fuzzing performance using principled techniques

3.1 Introduction

Greybox fuzzing has achieved much progress over the past few years, becoming more accepted in industry applications while receiving much attention in academia. Fuzzing's scalability and soundness have led security researchers to find a multitude of vulnerabilities in a wide variety of software, including IoT devices [19–21], Android apps [98], kernels [14–18], and application software [1, 38, 39, 44, 53].

Many state-of-the-art greybox fuzzers are based on American Fuzzy Lop (AFL) [1]. AFL is a classic mutation-based greybox fuzzer offering a versatile and robust architecture that allows developers to port its design to numerous platforms and operate on vastly different fuzzing targets. This has sparked interest in the research community, conceiving a number of AFL-derived fuzzers with numerous improvements [31, 38, 39, 43, 44, 53].

However, their respective strategies are limited by randomized algorithms. For example, AFL-

based fuzzers obtain program feedback in the form of branch coverage by recording the hit counts of each branch in a fixed-size bitmap called branch count table. Branches' IDs are determined randomly at static time to index the table. Randomly assigned IDs result in potential collisions where two branches correspond to the same ID, also known as the branch collision problem. On the other hand, the importance of context-sensitive branch counting can be corroborated by its extensive implementation in newer fuzzers [44, 53]. The increased unique branches brought by this new context information exacerbate branch collision problem.

An intuitive solution to mitigate this problem is to increase the branch count table's size, which is state-of-the-art fuzzers' approach. However, during our tests with programs such as *tcpdump*, the utilization rate of bitmaps can reach up to 36.6% even when enabling context-sensitivity using an enlarged 1MiB bitmap. As shown by Gan et al. [31], such utilization rates can induce very high collision rates, while an enlarged buffer reduces execution throughput by 30% on some programs. AFL++'s LTO mode statically assigns each branch a unique ID to achieve collision-free. However, its design does not accommodate for context-sensitivity, which is important for the fuzzer to detect subtle but important changes in a program's execution state.

Therefore, fuzzer developers have to face a trade-off between fine-grained but slow feedback or a fast but inaccurate one. Such trade-off has been carefully studied in [35]. Thus, there is a need for a better solution that takes a principled approach towards providing detailed, accurate, and efficient branch counting.

On the other hand, little effort is put into mutators. AFL-based fuzzers generally use heuristic methods, most of which are based on randomization. Even fuzzers with solvers have unrealistic assumptions, which often lead to failure and force the fuzzer to turn to randomization as a last resort. For example, in Angora, lots of "odd heuristics and parameters" [99] are added to the code. These heuristics caused uneven performances across trials. Therefore, [100] proposes a series of methods including repeated trials to guarantee the comparison is fair. However, real-world bugs are far and rare. Even ten repeated trials cannot guarantee a bug being found.

We carefully study the state-of-the-art fuzzers with embedded solvers and find these fuzzers

generally work in the following fashion. First, the fuzzer picks an unsatisfied branch predicate to solve. Then, it identifies the input sections that can affect the predicate's outcome through techniques such as dynamic taint analysis. Next, the fuzzer uses the solver to identify and exploit certain features of the predicate to solve it. The fuzzer continues to solve the target predicate until either the predicate is satisfied or the solver has exhausted its time budget. It then picks another predicate and repeats the process mentioned above. For instance, REDQUEEN attempts to identify and tackle checksums and hashes through techniques similar to magic byte matching, but it cannot solve general arithmetic predicates [43]. QSYM uses a modified concolic solver to solve the target predicate, but these solvers cannot solve constraints with complicated forms such as nonconvexity [88]. Angora converts the predicate to an objective function $f(\mathbf{x})$ to optimize using gradient descent, where $\mathbf{x}$ represents sections of input bytes [44]. Using numerical differentiation, Angora approximates the objective function's gradient and performs descent by mutating the corresponding input sections.

Some solvers fall back to random mutation when their assumptions do not hold for scenarios in real-world programs. Mathematical methods such as gradient descent are designed to work on functions in the real domain, which renders these solving methods ineffective against real-world constraints where many are in the bounded integer domain. Therefore, fuzzers that utilize these methods can only solve a subset of the predicates for the following reasons. 1) They believe the mutation amount $\Delta\mathbf{x}$ is *always* an integer, and 2) the predicate may overflow when the mutation amount derived from an integer $\Delta\mathbf{x}$ is too large. Therefore we need to find a way to allow solvers assuming real domain to work with branch predicates in real-world programs, allowing the fuzzer to release its full potential instead of rolling a dice and hoping for the best.

The fuzzing process involves two distinct tasks: exploration and exploitation. During the exploration phase, the fuzzer generates seeds to achieve broader coverage of the target codebase. Conversely, the exploitation phase focuses on uncovering bugs by generating seeds that lead to program crashes or other notable behaviors. While state-of-the-art fuzzers employ various strategies for exploration, many rely on heuristic-based methods for bug exploitation. For instance, popular

fuzzers like AFL and AFL++ employ random mutation techniques to blindly trigger bugs. In contrast, more advanced fuzzers like Angora identify specific instructions and attempt to insert values such as `NULL` or `INT_MAX` to trigger out-of-bounds memory access bugs. However, these approaches may fall short in detecting bugs that require non-trivial triggering conditions, such as specific input values or particular triggering mechanisms. Consequently, the effectiveness of the fuzzer may be compromised in such cases.

These problems are the current blocking issues when we hope to improve fuzzing effectiveness. A collision-prone and imprecise branch coverage feedback mechanism will cancel out the benefits of improved mutation methods, as the fuzzer would likely miss the resulting increased program states. A more sophisticated mutator cannot deliver its promise unless the fundamental assumptions hold under most circumstances. Ineffective exploitation methods render the fuzzer incapable of triggering bugs within code that has been already explored. We believe deterministic algorithms produce more consistent, predictable, reproducible results. Therefore, we wish to eliminate the randomness used in these two components. After re-evaluating these methods, we design techniques that address each aspect of the issues mentioned above:

First, we combine the best of two worlds by designing a branch coverage feedback mechanism that is collision-free *and* context-sensitive. We use static analysis to identify all possible branches present within the program. Instead of assigning each branch a static ID like current approaches, we give each branch a relocatable, function-local incremental ID. Additionally, we statically determine all possible *first-order* function contexts, i.e., function contexts are determined solely by the call site. For each function, we identify its direct call sites at static time. For indirect function calls, we assume any function with the same signature may be called at runtime. Thus, each branch's context-sensitive ID at runtime is determined by its function-local ID and the current function context. Furthermore, we develop an algorithm to remove unnecessary instrumentation while maintaining accuracy to reduce the table size. We also prove the correctness of the algorithm. To adapt to more extensive programs, we statically determine the required size for the branch counting table and negotiate a suitably-sized buffer automatically with the fuzzer at initialization. This

approach allows for more fine-grained feedback while reducing overhead, improving the fuzzer's ability to observe execution state changes in the program.

Next, we design a *compensated step* method that adapts solver algorithms developed for values in the real domain to integer domains, where many real-world programs run. To demonstrate the effectiveness of this approach, we use a gradient descent solver and apply our modifications. The high-level idea of this method is to clip the fractional values that could not be applied to integer values and *compensate* them to other components of the input vector. We denote the input vector as $\mathbf{x}$, the original mutation amount as $\Delta\mathbf{x} \in \mathbb{R}^n$, where n is the dimensions of the input vector, i.e., the number of input bytes of a predicate. Our target is to find a compensated mutation amount $\Delta\mathbf{x}' \in \mathbb{Z}^n$, such that $f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x} + \Delta\mathbf{x}')$. We also make some modifications to the original gradient descent solver such that *compensated mutation* can perform well in real-world situations. Specifically, we first modify the initial step size such that it is set to the smallest possible value by which the predicate can change, then doubling the step size value upon each successful descent step. We also used a different differential approach to get a more precise gradient.

Finally, we propose proactive exploitation that augments the fuzzer's bug detection capabilities. This method works by first identifying *exploit points*, i.e., locations where bugs may be present, in the target program during static analysis. We identify values according to its exploitation type that may trigger a bug, including divide by zero, out-of-bounds memory access, and memory allocation. In contrast with filling the input with interesting or randomized values, we utilize the solver to change the specific input values such that each exploit point will possibly trigger a bug during fuzzing. In order to maintain fuzzing throughput, we prioritize conventional exploration, i.e., we first solve as many branch predicates as possible, thus covering as much code as possible, then for all exploit points found in the covered code, we attempt to trigger bugs within the program. Additionally, we devise a way to lower the runtime costs due to the added instrumentation code based on the observation that most instrumented code during one execution of the fuzzed program does not require execution. We clone each function into copies that are instrumented with different instrumentation types, such as one copy for exploration feedback, another two for memory-related exploitation and integer-related

exploitation, respectively. At runtime, we only execute the corresponding instrumented function if it contains the target predicate or exploit point.

We implement a prototype fuzzer Valkyrie to deliver better performance through our improvements. We evaluate Valkyrie's effectiveness on standard dataset Magma and real-world programs. On Magma, Valkyrie found 21 unique integer and memory errors with no need for any randomization methods, 10.5% and 50% more than AFL++ and Angora, respectively. We also examine the performance of Valkyrie on real-world programs. First, our tests show that Valkyrie increased branch coverage by 8.2% compared with AFL++, and 12.4% compared with Angora. Second, we demonstrate that Valkyrie's branch counting mechanism allows for collision-free branch counting. At the same time, when using a bitmap with comparable size to Valkyrie's, AFL and Angora result in significant bitmap utilization rates, leading to high occurrences of collisions. Finally, we compared Valkyrie's solver with Angora's to show that even without any heuristics, our compensated step mutation can still do better than Angora.

This chapter makes the following contributions:

1. We propose a collision-free branch counting method and an algorithm to reduce branch count table size.

2. We propose an efficient mutation method for predicate solver. With the new solver, we can effectively target some memory and integer bugs during fuzzing.

3. We implement a prototype fuzzer Valkyrie using these deterministic techniques and evaluated its effectiveness and performance.

4. We demonstrate Valkyrie delivers a more stable and uniform performance than other commonly seen fuzzers on benchmarks and real-world programs.

3.2 Background and motivation

AFL is a classic mutation-based greybox fuzzer. AFL monitors the program state by inserting light instrumentation and monitoring branch coverage states. It then uses a series of heuristics and randomized methods to mutate existing seeds. The instrumented program is executed using the mutated seed. AFL will save the new seed if a new branch state is triggered.

Most fuzzers in the AFL family inherit these techniques with some modifications. For instance, fuzzers in the AFL family generally use a fixed-sized bitmap to record branch coverage information, allowing the fuzzer to identify new triggered states and save the mutated input as a seed for further mutation. During program execution, the instrumentation code increments the branch's bitmap entry whenever a new branch is executed. Some AFL-derived fuzzers implement context-sensitive branch counting [44, 53] to assist in discerning more unique states.

However, since the branch ID is determined randomly during instrumentation, it is not unique and can lead to branch collision. Gan et al. demonstrated that collisions are non-trivial and increase with the number of branches present within a program [31]. Paired with context-sensitivity, which significantly increases the number of unique branches observable by the fuzzer. Branch collisions pose a significant challenge when improving fuzzing effectiveness.

There are several attempts to mitigate the problem. For instance, Angora defaults to a larger bitmap size, which has been proved ineffective by Gan et al. since it does not eliminate collisions and slows down execution speed significantly. Gan et al. proposed CollAFL, which assigns IDs using non-random algorithms that greatly reduces collisions. AFL++ offers an optional *LTO mode* that provides collision-free branch counting [53]. However, the former cannot adjust to programs automatically, while the latter is experimental and buggy. Besides, both approaches lack context-sensitivity.

Fuzzers in the AFL family randomly mutate the entire input. Random mutation becomes somewhat ineffective after the "easy" branches are solved. More recent developments focus on using solvers to solve branch predicates to dive deeper into the code. It is guaranteed to alter the control

flow once the predicate is solved and possibly yield a new path. Many solvers have been proposed, including input-to-state-correspondence [43], concolic solvers [88], and gradient methods [44]. These fuzzers generally operate using the following workflow: 1) it identifies the corresponding input sections of the target predicate, 2) then it derives relevant properties of the predicate, such as the gradient, and 3) it mutates the input sections with its predicate solver using the above information.

However, these methods are limited in real-world scenarios. For example, the gradient method used in Angora assumes the input domain to be continuous when it is discrete in most cases. This limits its ability to solve many real-world predicates, which becomes difficult and almost impossible to solve using continuous domain assumptions. Listing 3.1 is an example code copied from libjpeg, where three input bytes are involved, two of which describes the buffer length and the other is the number of components in the buffer. There is a sanity check before the program consumes the buffer. Angora may convert the check into an objective function $f(\mathbf{x}) = |\mathbf{g}\mathbf{x}^T - 8|$ where $\mathbf{g} = [256, 1, -3]$ is the gradient. Then Angora tries to minimize it using classic gradient descent, where one can move input in arbitrarily small steps. Suppose the initial point is $\mathbf{x}_{init} = [0, 12, 1]$. When trying to take a small step $-\alpha\mathbf{g}$, say $\alpha = 0.1$, $-\alpha\mathbf{g} = [0, -0.1, 0.3]$, later two dimensions will find it unable to accept a fractional value and thus floored step to $[0, -1, 0]$ and result to $\mathbf{x} = [0, 11, 0]$. Angora would stagnate at this point. Since the first and the second dimensions are going in opposite directions, and all dimensions must be positive, Angora cannot find a next step.

One may argue that in this situation, we can use ceiling or rounding to solve this problem. However, we can always find code snippets where one operation works and the other two fail. The root cause is not clipping operations we choose to use, but that the assumption Angora made is not true in real-world programs, as each byte is bounded to $[0, 255]$ and the smallest step by which one input byte can change is either 1 or -1.

Bug exploitation in programs is another area that state-of-the-art fuzzers cannot perform as well as expected. In this chapter, exploitation refers to the process where the fuzzer tries to detect bugs from *exploit points* by generating a seed that crashes the program at the exploit point. State-of-the-art fuzzers generally employ heuristics-based bug exploitation methods. Popular fuzzers

```
1  static unsigned int NEXTBYTE (void);
2  static void process_SOFn (...) {
3      unsigned int length = (NEXTBYTE() << 8) + NEXTBYTE();
4      unsigned int num_components = NEXTBYTE();
5      if (length != 8 + num_components * 3)
6          ERREXIT("Bogus␣SOF␣marker␣length");
7      ...
8  }
```

Listing 3.1: Code snippet copied from libjpeg-9d. The program requires the length to be a specific amount to continue.

such as AFL and AFL++ generally use random mutation and, in some cases, dictionary values in an attempt to trigger bugs within the program. Angora only targets a small subset of possible exploitation points, for example, buffer indexing operations, and sets pre-defined values based on heuristics, such as `INT_MAX` or `NULL`. While these methods have been able to find numerous bugs in the history of these tools, more have been overlooked due to their non-trivial triggering conditions. Thus, we need a *proactive* bug exploitation method that allows the fuzzer to find these bugs within the explored program code.

3.3 Design

To overcome the limitations of state-of-the-art fuzzers, we propose the following improvements:

1. a branch counting mechanism that combines collision-free and context-sensitivity, with an instrumentation removal algorithm to reduce memory overhead while maintaining accuracy,
2. a predicate solver that adapts traditional optimization techniques designed for the real domain to bounded integer domain.
3. a bug triggering method based on our predicate solver.

3.3.1 Collision-free context-sensitive branch counting

Following the common practice in fuzzing, we record the visit counts of branches and use them to approximate the state of the program. We designed our mechanism to be both context-sensitive

and collision-free to improve the accuracy of the branch counting feedback. In current collision-free branch counting techniques, each branch is given a static unique ID b. Context-sensitive branch counting techniques generally use a context identifier c to differentiate between branches when appearing in different function contexts. Thus, we denote the tuple (c, b) as the context-sensitive branch. Our mechanism ensures that we record the visit count of each unique context-sensitive branch separately.

In contrast to AFL-derived branch counting mechanisms which use fixed-size branch counting tables, we wish to find the minimal space required for storing all the visit counts, allowing the fuzzer to adapt to any given program automatically. We achieve this in three steps. First, we identify all the unique context-sensitive branches. Then, in each function, we find the branches that don't need to be instrumented. Finally, for those branches that need instrumentation, we assign a unique sequential ID to each context-sensitive branch. This ID serves as the index of the branch in the branch-counting table.

3.3.1.1 Static branch edge ID generation

In contrast with AFL++'s approach of assigning a globally unique ID for each branch, we give each branch a relocatable function-local ID by visiting every function in the program, traversing the branch edges, and generating an incremental sequential ID statically for each branch. Then we collect the number of branches in each function. We also maintain an additional global *function offset* variable during runtime that is updated when calling or exiting a function. Thus the offset for each branch can be calculated by taking the function-local branch ID plus the function offset. Thus, we can dynamically calculate the unique branch ID using branch relocation depending on the specific function context.

3.3.1.2 Calculate the number of context-sensitive branches

Let F be the set of all functions in the program, $f \in F$ be a function, *branch_count*(f) be the number of branches in f, and *context_count*(f) be the number of different calling contexts of f.

Then the amount of branches is

$$n = \sum_{f \in F} \mathit{context\_count}(f) \cdot \mathit{branch\_count}(f)$$

We calculate $\mathit{branch\_count}(f)$ through the control flow graph of f. Calculating $\mathit{context\_count}(f)$ is more involved:

To avoid the explosion of the number of calling contexts (e.g., caused by recursion), we consider only one-level context, i.e., the context is determined by the call site only. Thus, we can determine explicit call sites easily.

3.3.1.3 Indirect function call context generation

To assign function context offsets for indirect function calls, we must identify all possible functions an indirect call site can call. To determine implicit call sites precisely, we would need precise points-to analysis. However, that is both difficult and expensive [101, 102].

Therefore, to find possible function contexts within a reasonable amount of time, we employ our method of approximating all candidate values of function pointers in indirect call sites. First, we determine the number of branch table entries that are required for each function in each context by taking the maximum number of branches of all functions. Then, we iterate over all function declarations in the program or library and classify them according to their function prototypes. Next, we find all operations that take the address of any function and add the respective functions to the candidate list. Finally, we find all candidate functions for each indirect function call site with the same function prototype. We reserve the amount of branch table entries required for each context. The function base offset is resolved at runtime by matching the actual pointer value with all possible candidate values.

3.3.1.4 Calculate the ID of each context-sensitive branch

Conceptually, for each function, we reserve a contiguous region of IDs that can store all the context-sensitive branches in the function.

To implement this, during instrumentation,

- In each function f
  - for each branch b, we sequentially assign a *function-local ID*, $ID(b)$, starting from 0.
  - for each potential call site c, we sequentially assign a context ID, $ID_f(c)$, starting from 0.
- We arbitrarily assign an order to all the functions, and assign an *ID offset*, $\mathit{offset}(f)$, to each function in the following way: for each function f_i, we set its offset $\mathit{offset}(f_i) = \mathit{offset}(f_{i-1}) + \mathit{context\_count}(f_{i-1}) \cdot \mathit{branch\_count}(f_{i-1})$. We initialize $\mathit{offset}(f_0)$ as 0.

At runtime, the ID of the context-sensitive branch (c, b) in function f is:

$$\mathit{offset}(f) + ID_f(c) \cdot \mathit{branch\_count}(f) + ID(b)$$

3.3.1.5 Redundant branch instrumentation removal

The benefit of instrumentation removal is twofold. First, it allows us to shrink the branch count table's size, reducing the memory overhead. Besides, branch counting is a time-consuming job where the program has to calculate the offset, fetch the entry, and save the result. If we could reduce the number of reported branches without affecting the distinguishability, then we can use the reduced branch counting to same runtime by not reporting these edges. To that end, we wish the path after instrumentation removal to be distinguishable from a different path after compression. Here, we formally define path and distinguishability:

Definition 3.3.1 (Path)**.** For a program with a CFG, the set of all edges are E. A complete path is a sequence of edges between basic blocks that represents one execution of a program. A compressed path is a subsequence of a complete path where only edges in $E' \subset E$ are kept.

Definition 3.3.2 (Distinguishability)**.** Suppose we have two complete paths P and Q, and their compressed paths P' and Q'. P' and Q' are said to be distinguishable when $P = Q$ if and only if $P' = Q'$.

We do not need to instrument an edge if whether it is taken does not distinguish two different paths. This introduced two requirements for our instrumentation removal. First, for each loop, at least one edge needs to be instrumented. Otherwise, we wouldn't distinguish how many times the loop has been executed. We use LLVM's definition of the loop[1] here and assume each loop has one and only one header block. Besides, for any basic block, exactly one of its outgoing edges needs no instrumentation. Because we can infer the status of that edge from other edges' status. For a basic block, if none of its instrumented outgoing edge is executed, then the only one that is not instrumented must be executed, and vice versa, if any instrumented edge is executed, then the edge without instrumentation is not executed. To satisfy both properties, we put labels on the edges before we instrument them. Algorithm 2 shows the algorithm.

For instance, in Figure 3.1, we only need to instrument and record the visit counts of branches `a`, `c` and `g` to sufficiently distinguish different paths. Our algorithm would work in the following fashion to achieve this result. Initially, all edges are labeled as *delete*. We iterate over all loops' header block(A and C) first and mark the loops' outgoing edges(`a` and `g`) as *keep*. Then for each basic block, we have exactly one outgoing edge labeled as *delete* and mark others as *keep*. Thus only `c` is kept. Notice that whether we keep `c` or `b` doesn't change the branch table's size, nor the distinguishability of the instrumentation. We will prove this property in Theorem 1. Finally, we instrument all edges marked as *keep*, including branches `a`, `c` and `g`.

Table 3.1 shows how paths are compressed after our instrumentation optimization. Column four of the table shows the branch counting table if no compression is used. Each edge needs a counter to record how many time it has been executed. On the other hand, with our path compression we observe that only three edge need instrumentation, shortening our table size from eight entries to three entries, saving memory usage. Because many edges doesn't have a corresponding counter,

[1] `https://llvm.org/docs/LoopTerminology.html`

they don't report their execution at runtime, lowering our runtime overhead. We find that the compression rate (column three) can be as high as 50%, that means we can save more than half of the runtime overhead. Finally, as we will prove in the following sections, the distinguishability is not affected by compression, which removes the necessity to convert the compressed path to the full path, since we can directly compare the compressed path and differentiate two execution traces.

We formally prove the algorithm's correctness:

Theorem 1. *Let P and Q be two paths. Let E be the set of all edges in the CFG, and $E' \subset E$ be the set of edges kept by Algorithm 2. Let P' and Q' be the compressed path of P and Q, respectively, generated from E'. Then $P = Q$ if and only if $P' = Q'$.*

Proof. **Necessity** (the right direction): Since $P = Q$, their subsequence on E' must also be equal.

Sufficiency (the left direction): Prove by contradiction. Assume $P \neq Q$ but $P' = Q'$. Let $P = (A, p_1, \cdots)$, $Q = (A, q_1, \cdots)$, where A is the longest common prefix of P and Q. Therefore, p_1 and q_1 are different but they start from the same basic block B, so B must have $n > 1$ outgoing edges. Algorithm 9 to Algorithm 15 of Algorithm 2 guarantees that at least $n - 1$ of the edges are marked *keep*, so at least one of p_1 and q_1 is marked as *keep*.

If both p_1 and q_1 are marked as *keep*, then they both appear in E', so $P' = (A', p_1, \cdots)$ and $Q' = (A', q_1, \cdots)$ where A' is the compressed path of A. Since $p_1 \neq q_1$, $P' \neq Q'$, but this contradicts our assumption.

If only one of p_1 and q_1 are marked as *keep*. Without loss of generality, let p_1 be marked as *keep*. So $P' = (A', p_1, \cdots)$. The assumption $P' = Q'$ implies that $Q = (A, q_1, B, p_1, \cdots)$, i.e., Q contains a cycle (q_1, B) and no edge in the cycle is marked as *keep*. But Algorithm 3 Algorithm 8 of Algorithm 2 prevented this. □

3.3.2 Compensated mutation assisted solver

While random mutation operators generally used by the AFL family of fuzzers can quickly solve "easy" predicates, predicates with a small feasible input space are difficult for them to solve, especially

Algorithm 2 Procedure for determining which branches to instrumentation in a function.

```
1: function FindEdgesToInstrument(CFG)
2:     Mark all edges as delete.
3:     for Loop l ∈ CFG do
4:         h ← l's header block
5:         for Edge e = (h, b) ∈ h's outgoing edge do
6:             Mark edge (h, b) as keep.
7:         end for
8:     end for
9:     for Block b ∈ CFG do
10:        E = set of all outgoing edges of b
11:        if ∃e1 ≠ e2 ∈ E, both are markded as delete then
12:            ∀e ∈ E, mark e as keep
13:            mark e1 as delete
14:        end if
15:    end for
16:    Instrument all edges marked with keep
17: end function
```

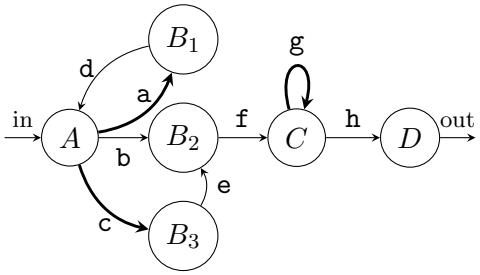


Figure 3.1: Examples of branches that do not require instrumentation. Only thickened edges need instrumenting.

when the predicate is an equality comparison. In Listing 3.1, `num\_components` has 256 possibilities, thus 256 possible inputs to satisfy the comparasion. However, there are 256^3 possible inputs for three bytes, making it difficult for fuzzers that randomly generate inputs. On the other hand, even state-of-the-art fuzzers with a solver may fail because their assumptions are not true.

Therefore, we need a new solver that properly handles the bounded integer domain that largely exists in real-world programs.

We use the notation $f(\mathbf{x})$ to represent its objective function for each predicate. $\mathbf{x}$ is a vector determined by a subset of the input bytes. The fuzzer maps input bytes to $\mathbf{x}$ using dynamic taint analysis tools like DataFlowSanitizer [80]. The range of each dimension of $\mathbf{x}$ is determined by its type, bit width, and signs, which Valkyrie computes by static analysis. For simplicity, we refer to

Table 3.1: Examples of path compressions in Figure 3.1. Grey areas in column five means that we didn't allocate memory for thoese edges. Notice how edge `f` and `h` *must* be executred, thus there is no need to instrument them.

Path	Compressed path	Compression rate	Uncompressed Matcher table								Compressed Matcher table							
			a	b	c	d	e	f	g	h	a	b	c	d	e	f	g	h
`bfh`	-	100%		1				1		1								
`cefh`	`c`	75%			1		1	1		1			1					
`adbfgh`	`ag`	66%	1	1		1		1	1	1	1						1	
`cefggh`	`cgg`	50%			1		1	1	2	1			1				2	
`adadadbfggh`	`aaagg`	55%	3	1		3		1	2	1	3						2	
`adcefgh`	`acg`	57%	1		1	1	1	1	1	1	1		1				1	
`adcefggggh`	`acgggg`	40%	1		1	1	1	1	4	1	1		1				4	

Table 3.2: Conversion table between branch predicate expressions, their corresponding objective functions and solver targets. δ represents the smallest possible positive value that the numerical type can represent. For integers, $\delta = 1$.

Predicate	Objective	Angora's constraint	Valkyrie's constraint
$a > b$	$f = b - a$	$f < 0$	$f < 0$
$a < b$	$f = a - b$	$f < 0$	$f < 0$
$a = b$	$f = a - b$	$\lvert f \rvert \leq 0$	$f = 0$
$a \geq b$	$f = b - a - \delta$	$f < 0$	$f < 0$
$a \leq b$	$f = a - b - \delta$	$f < 0$	$f < 0$
$a \neq b$	$f = a - b$	$-\lvert f \rvert < 0$	$f < 0$ or $f > 0$

the maximum and minimum value that can be represented by $\mathbf{x}_i$ as min_i and max_i.

f is a blackbox function determined by the predicate, as shown in Table 3.2. When the predicate becomes unreachable because a new input alters the program path, we set $f(\mathbf{x})$ to a value that violates the objective. For example, when the objective is $f(\mathbf{x}) < 0$, then we set $f(\mathbf{x}) = +\infty$.

The effectiveness of state-of-the-art predicate-solving fuzzers implies that many predicates in the program are solvable using principled methods. For example, Angora assumes that the objective functions of predicates are continuous, therefore it uses a gradient-descent-derived solver. However, program inputs usually take the form of byte values that are bounded and discrete. Therefore, solvers developed with a continuous range assumption require modifications to adapt to real-world

situations.

We design a compensated mutation technique that mitigates this problem. The main idea of compensated mutation is when given a target step $\Delta\mathbf{x} \in \mathrm{R}^n$ that the solver wants to apply to the input, we find a $\Delta\mathbf{x}' \in \mathbb{Z}^n$ such that $f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x} + \Delta\mathbf{x}')$. To do this, we clip the fractional values that could not be applied to integer values and *compensate* them to other components of the input vector. To demonstrate how this approach works and its effectiveness, we apply this technique to a gradient descent solver, albeit with some modifications.

3.3.2.1 Compensation from real domain to integer domain

Current methods resort to integer flooring when given a vector of fractional numbers $\Delta\mathbf{x} \in \mathbb{R}^n$ to apply to a vector of integer numbers. However, we cannot guarantee that the floored value $\lfloor\Delta\mathbf{x}\rfloor$ will result in a similar function value, especially when components have large coefficients in the function. To avoid precision loss due to rounding techniques of any kind, we wish to find an integer vector $\Delta\mathbf{x}' \in \mathbb{Z}^n$ such that $f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}+\Delta\mathbf{x}')$. The main idea behind the *compensated step* is that for a $\Delta\mathbf{x}$ as well as its *gradient* on the function, we traverse through each component, apply a suitable integer mutation value, and *compensate* the fractional values that were not applied into other components. We denote $\mathbf{r}_i$ as the amount that we intend to add to $\mathbf{x}_i$ and $\Delta\mathbf{x}'_i$ for the actual integer value that is added. The difference between $\mathbf{r}_i$ and $\Delta\mathbf{x}'_i$ is the value that needs to be compensated to another component of the input vector. We call this difference *carry amount* and use notation $\mathbf{c}_i$. Thus we have:

$$\mathbf{c}_i = \mathbf{r}_i - \Delta\mathbf{x}'_i$$

However, in many cases, it is not possible for the last dimension to take fractional values or reach its upper bound. To ensure that the function value remains the same, we introduce the concept of a "carry amount" ($\mathbf{c}_{i-1}$) to be added to the next dimension. If we applying the carry amount ($\mathbf{c}_i$) to the i^{th} component, the objective function value should change by $\mathbf{c}_i\mathbf{g}_i$, where $\mathbf{g}_i$ represents

the partial derivative of dimension i. However, since $\mathbf{c}_i$ is a fractional value that cannot be directly applied, we need to carry this amount over to dimension j. In order to maintain the same change in the function value, we should add another term of $\frac{\mathbf{c}_i \mathbf{g}_i}{\mathbf{g}_j}$ to the original value $\mathbf{x}_j$. As a result, the representation $\mathbf{r}_i$ of the i^{th} dimension consists of two parts. The first part is the original value $\mathbf{x}_i$, and the second part is the carry amount from the last dimension, which is $\frac{\mathbf{c}i-1\mathbf{g}i-1}{\mathbf{g}_i}$. We can write the compensation process in Equation 3.1:

$$\begin{aligned} \mathbf{c}_0 &= 0 \\ \mathbf{r}_1 &= \Delta\mathbf{x}_1 \\ \mathbf{r}_i &= \Delta\mathbf{x}_i + \frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i} \\ \mathbf{c}_i &= \mathbf{r}_i - \Delta\mathbf{x'}_i \end{aligned} \tag{3.1}$$

Finally, to obtain the integer value $\Delta\mathbf{x}'_i$, most of the time we use $\Delta\mathbf{x'}_i = \lfloor \mathbf{r}_i \rfloor$. This is different than $\lfloor \Delta\mathbf{x}_i \rfloor$. As shown in Equation 3.1, $\mathbf{r}_i$ is the sum of the target value $\Delta\mathbf{x}_i$ and the amount carried over from the previous component $\mathbf{c}_{i-1}$ corrected by the fraction of gradients $\frac{\mathbf{g}_{i-1}}{\mathbf{g}_i}$. There are few exceptions where we don't floor $\mathbf{r}_i$:

1. $\mathbf{x}_i + \lfloor \mathbf{r}_i \rfloor > max_i$. This means we could overflow this dimension, thus we set $\Delta\mathbf{x'}_i = max_i - \mathbf{x}_i$.
2. $\mathbf{x}_i + \lfloor \mathbf{r}_i \rfloor < min_i$. Similarly, we set $\Delta\mathbf{x'}_i = min_i - \mathbf{x}_i$.
3. The carry amount $\mathbf{c}_i$ is so large that all the dimensions will be overflown by it. In this case we try $\Delta\mathbf{x'}_i = \lceil \mathbf{r}_i \rceil$.

It is not hard to derive the following relation using calculus and Equation 3.1:

$$
\begin{aligned}
f(\mathbf{x}+\Delta\mathbf{x}') &\approx f(\mathbf{x}) + \mathbf{g}^T\Delta\mathbf{x}' \\
&= f(\mathbf{x}) + \sum_i \mathbf{g}_i(\mathbf{r}_i - \mathbf{c}_i) \\
&= f(\mathbf{x}) + \sum_i [(\mathbf{g}_i \cdot \Delta\mathbf{x}_i + \mathbf{g}_i \cdot \frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i}) - \mathbf{g}_i\mathbf{c}_i] \\
&= f(\mathbf{x}) + \sum_i \Delta\mathbf{x}_i\mathbf{g}_i - \mathbf{g}_n\mathbf{c}_n \\
&= f(\mathbf{x}+\Delta\mathbf{x}) - \mathbf{g}_n\mathbf{c}_n
\end{aligned}
\tag{3.2}
$$

Therefore, the loss of our method can be as low as $|\mathbf{g}_n\mathbf{c}_n|$. In practice, we use a permutation matrix to sort the components in the descending order of the absolute value of their gradients for the following reasons:

1. Since in most cases $\mathbf{c}_{i-1} < 1$, we need $\frac{\mathbf{g}_{i-1}}{\mathbf{g}_i} > 1$, otherwise the compensation won't affect $\mathbf{r}_i$ too much.

2. We also want $\frac{\mathbf{g}_{i-1}}{\mathbf{g}_i}$ to be as small as possible, so it would not amplify $\mathbf{r}_i$ too much that we have to push $\mathbf{x}'_i$ to its bound.

3. As shown in Equation 3.2, a small $|\mathbf{g}_n|$ would reduce the error incurred by compensated step.

The whole process is described in Algorithm 3. First, we sort the inputs based on the gradient. Then we calculate $\mathbf{r}_i$ for each dimension based on Equation 3.1. We then choose $\Delta\mathbf{x}'$ based on $\mathbf{r}_i$ as described before.

This method is applicable to any solver that can obtain the *gradient* of each input component. The gradient can be obtained using a variety of methods, such as using white-box analysis and receiving an explicit expression, or through numerical methods to approximate the gradient. In our approach, we use a numerical estimation. In the following section, we describe our modifications for improved numerical differentiation in real-world fuzzing scenarios in the following part.

Algorithm 3 Compensated step

function CompensatedStep($\mathbf{x} \in \mathbb{Z}^n, \Delta\mathbf{x}, \mathbf{g} \in \mathbb{R}^n$)
  $P \leftarrow$ Permutation matrix s.t. $\forall i < j, |P\mathbf{g}_i| \geq |P\mathbf{g}_j|$
  $\mathbf{x} \leftarrow P\mathbf{x}$
  $\Delta\mathbf{x} \leftarrow P\Delta\mathbf{x}$
  $g \leftarrow Pg$ ▷ Sort dimensions in the descending order of the absolute value of the gradient
  $\mathbf{c}_0 \leftarrow 0, \mathbf{g}_0 \leftarrow 1$
  for i in 1..n **do**
    $\mathbf{r}_i \leftarrow \Delta\mathbf{x}_i + \frac{\mathbf{c}_{i-1}\mathbf{g}_{i-1}}{\mathbf{g}_i}$
    $\Delta\mathbf{x}'_i = \lfloor \mathbf{r}_i \rfloor$
    if $\mathbf{x}_i + \Delta\mathbf{x}'_i > max_i$ **then**
      $\Delta\mathbf{x}'_i = max_i - \mathbf{x}_i$
    else if $\mathbf{x}_i + \Delta\mathbf{x}'_i < max_i$ **then**
      $\Delta\mathbf{x}'_i = min_i - \mathbf{x}_i$
    else if $\mathbf{r}_i - \Delta\mathbf{x}'_i$ is too large for the rest dimensions **then**
      $\Delta\mathbf{x}'_i = \lceil \mathbf{r}_i \rceil$
    end if
    $\mathbf{c}_i = \mathbf{r}_i - \Delta\mathbf{x}'_i$
  end for
  return $P^{-1}\Delta\mathbf{x}'$
end function

3.3.2.2 Compensated gradient descent

With the compensated step, here we modify the traditional gradient descent solver to tackle real-world scenarios. Although compensated step can be applied to any solvers, we find gradient descent better suited for our needs. Compensated step heavily rely on a gradient to work, which is the same for gradient descent.

Modified differentiation for a more accurate gradient. Since the predicates' mathematical expressions are unknown and we treat them as black-box functions, we cannot derive a gradient symbolically. However, the traditional differentiation method lack accuracy since a valid gradient's absolute value may be less than 1. For example $x = 5, f(x) = \lfloor x/4 \rfloor$, where flooring the result is the semantic of integer division in C programs. In this case, we find $f(x+1) = f(x) = f(x-1) = 1$ and end up with zero gradient. We need an approximated gradient instead of a zero gradient to keep the algorithm going.

Therefore, to obtain the partial gradients of a particular predicate, we use a modified numerical differentiation method on each dimension to derive a partial gradient. When calculating differentiation for dimension i, we create a unit vector $\mathbf{e_i} \in \mathbb{R}^n$ where only the i-th element is 1 and all other

elements is 0. We add and subtract $\mathbf{x}$ with this $\mathbf{e_i}$ and observe f's value change to derive a gradient.

Furthermore, we introduce amplifiers β_+ and β_- to increase the unit step size. β_+ and β_- starts with 1. We keep doubling β_+ and β_- until we find a non-zero $f(\mathbf{x}+\beta_+\mathbf{e_i})-f(\mathbf{x})$ or $f(\mathbf{x})-f(\mathbf{x}-\beta_-\mathbf{e_i})$. Then we can compute the gradient in the i-th dimension, $\mathbf{g}_i$, using Equation 3.3:

$$\mathbf{g}_i = \frac{f(\mathbf{x}+\beta_+\mathbf{e_i}) - f(\mathbf{x}-\beta_-\mathbf{e_i})}{\beta_+ + \beta_-} \tag{3.3}$$

If the amplifier β grows very significant without finding a practical value, we consider the gradient to be zero. β is considered large if $\beta > \frac{1}{2}(max_i - min_i)$. If both directions turn out to be zero, we assume this direction to have zero gradient. By repeating this process on all dimensions, we get a differentiation vector $\mathbf{g}$.

Determine the step size in descent. In the state-of-the-art solver, it takes a step $\Delta\mathbf{x} = -\alpha\mathbf{g}$ to descend in each iteration. However, it is challenging to set α. If we set it too small, $\mathbf{x}$ may move slowly or even stagnate. For example, $f(x) = \lfloor x/4 \rfloor$, if we move x by 1, $f(x)$ will not change. But if we set it too large, we may overshoot, causing the function to descend more than intended.

Therefore, we take the advantage of the fact that given a small step $\Delta\mathbf{x}$, f is approximately linear. There is an ϵ ball $B_\epsilon(\mathbf{x})$ such that for small enough $\epsilon \in \mathbb{R}$ such that given $||\Delta\mathbf{x}||_\infty < \epsilon$, $\mathbf{g}$ being the gradient, we have

$$f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{g}^T\Delta\mathbf{x}$$

We select an α such that $f(\mathbf{x})$ will change approximately by the smallest possible increments or decrements.

$$\begin{aligned} v &= \max(1, \min_{\mathbf{g}_k \neq 0}(|\mathbf{g}_k|)) \\ \alpha &= \frac{v}{\mathbf{g}^T\mathbf{g}} \end{aligned} \tag{3.4}$$

If v is small, $f(\mathbf{x}-\alpha\mathbf{g}) - f(\mathbf{x}) \approx -v$ by Equation 3.2. We introduced a minimum non-zero gradient $\mathbf{g}_k$ because if $|\mathbf{g}_k| > 1$, the minimal change possible to $f(\mathbf{x})$ is $|\mathbf{g}_k|$ instead of 1, since $f(\mathbf{x})$ is a

Algorithm 4 Descent routine

Require: f
1: **function** DESCENT($\mathbf{x}, \mathbf{g} \in \mathbb{R}^n$)
2: $v \leftarrow max(1, \min_{i \text{ s.t.} \mathbf{g}_i \neq 0} |\mathbf{g}_i|)$
3: $\alpha \leftarrow v/\mathbf{g}^T\mathbf{g}$
4: $\mathbf{x}_{prev} \leftarrow \mathbf{x}$, $f_{prev} = f(\mathbf{x}_{prev})$,
5: **loop**
6: $\Delta\mathbf{x}' \leftarrow$COMPENSATEDSTEP($\mathbf{x}_{curr}, -\alpha\mathbf{g}, \mathbf{g}$)
7: $\mathbf{x}_{curr} \leftarrow \mathbf{x}_{curr} + \Delta\mathbf{x}'$, $f_{curr} = f(\mathbf{x}_{curr})$
8: **if** $f_{curr} = \infty$ **or** $|f_{prev}| \leq |f_{curr}|$ **then** ▷ Next step doesn't exist or the function is not descending.
9: **return** $\mathbf{x}_{prev}$
10: **else if** ISSOLVED(f_{curr}) **then**
11: **return** $\mathbf{x}_{curr}$
12: **end if**
13: $\alpha \leftarrow 2\alpha$, $\mathbf{x}_{prev} \leftarrow \mathbf{x}_{curr}$, $f_{prev} \leftarrow f_{curr}$
14: **end loop**
15: **end function**

discrete function. In each iteration, we double the step size to descend quicker. We revert the descent parameters to the initial state when we can no longer descend.

For non-linear functions, we could recalculate gradient in every step. However, since each dimension's gradient calculation requires us run the program under test a few times, calculating gradient is very expensive. We recognize that there are three possible outcomes for non-linear predicates. First, the execution path changes and the predicate is unreachable. In this case, we have no choice but to stop descending and use the value from the previous step as a result, a new gradient will be calculated later. Secondly, the function value may drop less than expected or even increase. We test if the new function value is still descending; if not, we return the previous step and recalculate the gradient. Finally, the function value may drop more than expected. Since our goal is to do gradient descent instead of keeping the function linear, we are fine with this step and keep going until we run into previous two cases.

The overall modified gradient algorithm is listed in Algorithm 4. We start by calculating the step size using Equation 3.4. Then we would decide whether to ascend or descend based on the current status of the function. Once the actual step $\Delta\mathbf{x}$ is determined, we calculate the compensated step using Algorithm 3. Finally we apply the integer step.

3.3.2.3 Solving motivating example

In the case of the example in Listing 3.1, we first formalize it as "given $f(\mathbf{x}) = \mathbf{g}^T\mathbf{x} - 8$, find $\mathbf{x}_{eq}$, s.t. $f(\mathbf{x}_{eq}) = 0$ " Suppose the input has been sorted by gradient, thus $\mathbf{g} = [256, -3, 1]$ and the initial point is $\mathbf{x}_{init} = [0, 1, 13]$, $f(\mathbf{x}_{init}) = 2$.

We start with $v = 1, \alpha = \frac{v}{\mathbf{g}^T\mathbf{g}}$, i.e. we try to decrease function's value by only 1. The first dimension will have $\mathbf{r}_1 = \Delta\mathbf{x}_1 = -\frac{\mathbf{g}_1}{\mathbf{g}^T\mathbf{g}}$. We find $\mathbf{x}_1$ is already 0 and cannot decrease more.

Thus we carry all the $\mathbf{r}_1$ to the next dimension, i.e. $\mathbf{c}_1 = \mathbf{r}_1 = -\frac{\mathbf{g}_1}{\mathbf{g}^T\mathbf{g}}, \Delta\mathbf{x}'_1 = 0$.

$\mathbf{c}_1$ is then applied to the next dimension, thus we have $\mathbf{r}_2$:

$$\begin{aligned}
\mathbf{r}_2 &= \Delta\mathbf{x}_2 + \frac{\mathbf{c}_1\mathbf{g}_1}{\mathbf{g}_2} \\
&= -\frac{\mathbf{g}_2}{\mathbf{g}^T\mathbf{g}} - \frac{\mathbf{g}_1}{\mathbf{g}^T\mathbf{g}}\frac{\mathbf{g}_1}{\mathbf{g}_2} \\
&= -\frac{1}{\mathbf{g}_2}\frac{1}{\mathbf{g}^T\mathbf{g}}(\mathbf{g}_1^2 + \mathbf{g}_2^2) \\
&= -\frac{1}{\mathbf{g}_2}\frac{1}{\mathbf{g}^T\mathbf{g}}(\mathbf{g}^T\mathbf{g} - \mathbf{g}_3^2) \\
&= \frac{1}{3}(1 - \frac{1}{\mathbf{g}^T\mathbf{g}})
\end{aligned}$$

$\mathbf{r}_2$ is again floored to 0, leaving $\mathbf{c}_2 = \mathbf{r}_2, \Delta\mathbf{x}'_2 = 0$.

Interestingly, we have

$$\begin{aligned}
\mathbf{r}_3 &= \Delta\mathbf{x}_3 + \frac{\mathbf{c}_2\mathbf{g}_2}{\mathbf{g}_3} \\
&= -\frac{1}{\mathbf{g}^T\mathbf{g}} + \frac{\mathbf{g}_2}{\mathbf{g}_3}\frac{1}{3}(1 - \frac{1}{\mathbf{g}^T\mathbf{g}}) \\
&= -\frac{1}{\mathbf{g}^T\mathbf{g}} - (1 - \frac{1}{\mathbf{g}^T\mathbf{g}}) \\
&= -1
\end{aligned}$$

Therefore $\Delta\mathbf{x}'_3 = -1$ and we end up with $\Delta\mathbf{x}' = [0, 0, -1]$. This would give us $\mathbf{x} = [0, 1, 12]$, $f(\mathbf{x}) = 1$. Since the descent is successful, we would double the step size, i.e. set $v = 2$ and descent again. Following a similar process would give us $\mathbf{x} = [0, 1, 10]$, $f(\mathbf{x}) = -1$. Because the absolute value is not descending, we would abort the descent instead of taking the step. We calculate the

gradient again and restart using $v = 1$. The final step would give us $\mathbf{x}_{eq} = [0, 1, 11]$, $f(\mathbf{x}_{eq}) = 0$.

3.3.3 Proactive bug exploitation

Current exploitation methods employed by state-of-the-art fuzzers such as AFL++ are merely best efforts, which are generally based on heuristics and magic byte insertions. These methods are ineffective when exploiting bugs that require non-trivial triggering conditions. Instead, we extend the predicate solver which is designed for program exploration into the domain of bug exploitation. We identify and designate *exploit predicates*, which are possible exploit points transformed into a predicate that the solver can handle. When exploit predicates are solved, they trigger a bug instead of explore a new path. The proactive bug exploitation process is divided into exploit point identification through static analysis and exploit predicate solving during fuzzing.

3.3.3.1 Exploit point identification

During static analysis, we identify susceptible instructions that have a probability of triggering a crash as *exploitation points*. For each possible exploitation site, we identify an *exploitable value*, i.e., a value that will trigger a crash at this point, where we instrument predicates for the solver to try triggering these bugs. Here, we select three exploitation cases where an architecture failure will be triggered:

Divide by zero. The relevant susceptible instruction takes the form of a division operation, specifically `result = dividend / divisor`. We instrument a predicate $divisor == 0$ so that the solver will try to move the divisor to zero. In practice, we find programmers will most likely check if divisor is zero, while forgetting the possibility that the divisor can *overflow* to zero. Therefore, we instrument another predicate $divisor == \text{MAX} + 1$.

Memory indexing. If the memory index is larger than the size of the buffer, there may be a buffer overflow. Although buffer size is hardly known at runtime, all we need is to guide the solver to push the index to a higher value. The solver either finds it impossible due to index checks in the program, or trigger a buffer overflow. For each indexing, we instrument a predicate $\text{idx} > \text{MAX}$.

Memory allocation. If the size of memory allocation is not sanitized properly, a memory corruption can happen. There are two possibilities. If the allocated memory is less than desired, then it may result in a future buffer overflow, which may happen when the argument overflows to a small value. For example, `malloc((uint8_t)(257))` only allocates 1 byte of memory instead of 257 due to integer overflow. On the other hand, large allocation size requests may result in resource exhaustion, leading to the program being killed or returning a null pointer that may not be properly sanitized. This can happen when malformed or malicious inputs are processed without proper checks within the program or an integer underflow, for example `malloc(10 - 12)`. Therefore, we instrument two predicates: size $>$ MAX and size < 0. One predicate targets a very large allocation size while the other targets an integer underflow.

3.3.3.2 Exploration prioritized scheduling

We observe that if the buggy code cannot be reached by the fuzzer, then the bug cannot be triggered regardless of the resources spent on exploitation. Therefore, we prioritize exploration over exploitation so that we will have a better chance of triggering bugs. In each round of fuzzing, we always try to solve exploration predicates first, we only start trying exploitation predicates after we exausted exploration predicates. We achieve this by always setting exploitation predicates with lower initial priority. Since state-of-the-art fuzzers are using priority queue to do scheduling, our approach brings no overhead to the fuzzer. What's more, this approach guarantees that the solver will attempt to solve exploration predicates before proceeding to trigger exploit points. In our experience fuzzing Magma dataset (Section 3.4.1), the solver will attempt on all exploit predicates at least once except for a few trials.

On the other hand, we realize that most exploit predicates are infeasible since each one of them represents a bug. Therefore, spending too many resources on exploitation in a fuzzing campaign is unwise. Because our solver is deterministic, we will discard an exploit predicate after one attempt, as we have no reason to believe the second attempt will work. One exception is that we may find a predicate with different initial points. Since some predicates can be non-convex, the solver will try

to solve the same predicate with different initial points.

3.4 Evaluation

We are interested to know how well Valkyrie works in practice. We implemented Valkyrie to conduct a series of experiments to analyze the effectiveness of the entire fuzzer and individual components. We borrowed from Angora the dynamic taint tracking framework and instrumentation base code. We used the LLVM compiler framework for program analysis and instrumentation. However, the branch counting algorithm and the solver are independent of Angora's. The implementation of our branch counting mechanism uses *gllvm* [103] to consolidate the program's compiled LLVM IR into one module, allowing for full-program analysis. We have open-sourced Valkyrie, including all docker images and seeds used in evaluation to Github: https://github.com/organizations/ValkyrieFuzzer.

We propose the following research questions to help us understand the results and implications of our designs:

- **RQ1**: Is Valkyrie state-of-the-art? How does it fare on benchmarks such as Magma?
- **RQ2**: How does Valkyrie perform against similar fuzzers on real-world open-source programs?
- **RQ3**: Is our branch counting mechanism a better trade-off than that of AFL++ or Angora?
- **RQ4**: Is the solver assisted with compensated step better?
- **RQ5**: Can branch counting and solver contribute to bug finding in real world applications?

To answer these questions, we designed experiments to examine Valkyrie's performance on Magma and a select group of open-source programs. We then conducted two close examinations to address the latter two questions adequately.

First, we test Valkyrie on benchmark Magma v1.1 [52], then on real-world programs. We intend to test Valkyrie on a more robust benchmark FuzzBench [104], but Angora is not provided in the benchmark. The reason is that FuzzBench only allows programs to be compiled once, but Angora

Table 3.3: The list of fuzzers we used in our evaluation. Included are their respective versions and the arguments we provided to invoke the fuzzer.

Package	Version	Arguments
afl	2.57b	`-m 2048 -t 1000+`
MoptAFL	commit `339a21e`	`-m 2048 -t 1000+`
aflplusplus	3.01a	`-m 2048 -t 1000+`
angora	commit `3cedcac`	`-M 2048 -T 1`

Table 3.4: The list of projects we used in our evaluation. Included are their respective versions, the binary we used and the arguments we provided to invoke the binaries.

Package	Version	Program	Arguments
libjpeg-ijg	v9d	*cjpeg*	@@
jasper	2.0.12	*imginfo*	-f @@
jhead	3.04	*jhead*	@@
binutils	2.35	*nm*	-C @@
binutils	2.35	*objdump*	-x @@
xpdf	4.00	*pdftotext*	@@
binutils	2.35	*readelf*	-a @@
binutils	2.35	*size*	@@
libpcap / tcpdump	1.9.1 / 4.9.3	*tcpdump*	-e -vv -nr @@
libxml	2.9.10	*xmllint*	@@

requires two compilations to generate two versions of binaries. For fairness of the testing, we borrow the framework from Unifuzz [105] to test real-world programs. Each fuzzer runs in a containerized environment with *one core*. Each experiment lasted 24 hours and was repeated ten times, as suggested by [52]. In both experiments, we select AFL, AFL++, and Angora for comparison. We choose AFL as the reference fuzzer since it is a source of inspiration for many others. We also include AFL++, which has merged many improvements and function enhancements developed for AFL. We enabled llvm_mode, with AFLfast's power scheduling [38], MOpt's mutator [39], and non-colliding branch counting for AFL++. Angora is also a solver-based fuzzer with similar design goals to Valkyrie. We intend to compare to one of Angora's successors Matryoshka [85]. However, the tool is not available to us. Table 3.3 shows the the fuzzers' versions and arguments, Table 3.4 shows the versions and arguments of targets we fuzzed.

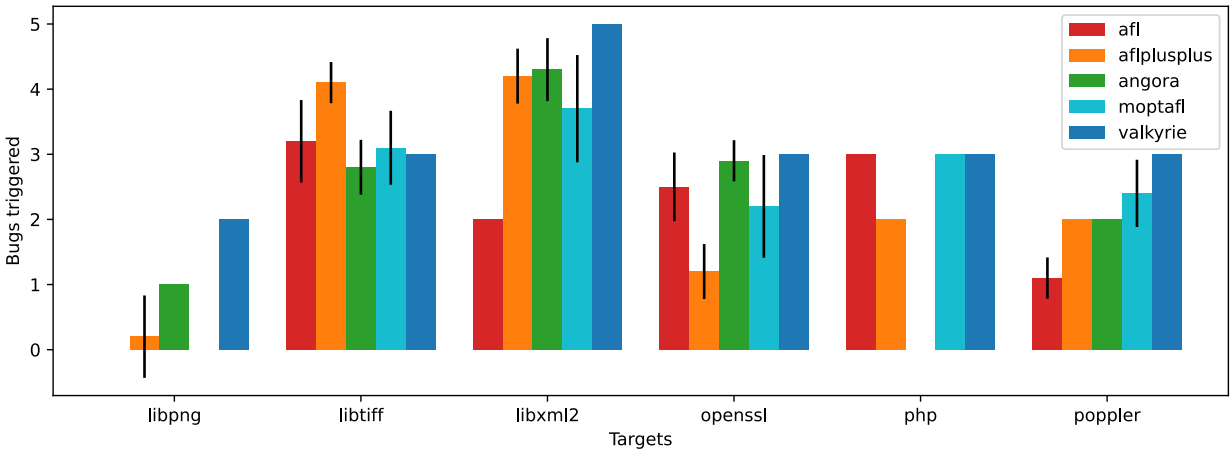


Figure 3.2: Arithmetic mean of number of integer and memory bugs triggered per trial per day. The black line shows 95% confidence interval. Valkyrie's performance is the same across ten trials.

3.4.1 Magma benchmark

To test whether Valkyrie is state-of-the-art, we would like to work on a benchmark with ground truth first. We examined Valkyrie's performance against other popular fuzzers on Magma v1.1 [52]. Magma is a collection of targets with real-world environments. It contains seven libraries and 16 binaries. Magma manually forward-ported these bugs in older versions to the latest versions. Unlike LAVA-M [106] where all bugs are synthetic and magic byte comparison, Magma has a spectrum of bugs covering most categories in Common Weakness Enumeration (CWE). Magma contains 118 bugs in total. There are 15 integer errors, six of which are divide-by-zero, and 58 memory overflows. The rest 45 bugs include use-after-free, double-free, 0-pointer dereference, etc.

However, Angora is a coverage-guided fuzzer that isn't designed to trigger bugs. We borrow ideas from [73, 107], for each potential bug, e.g. buffer overflow, we would insert a branch `if (ptr > buf_len)report();` so that Angora can see and solve the predicate. Therefore, for a fair comparison, we only tested on 15 integer errors and 58 memory bugs that can be converted to a predicate.

MoptAFL is also reported to be the best in the benchmark [52], therefore we included MoptAFL in this evaluation. We used the version provided in the benchmark. We want to see how Valkyrie compares with the state-of-the-art fuzzers.

Table 3.5: Average time used to trigger a bug in Magma. Bolded text shows the fastest to trigger a bug.

Bug ID	Valkyrie	angora	aflplusplus	moptafl	afl
AAH037	**15s**	15s	39s	20s	20s
AAH041	**15s**	15s	1m	33s	21s
JCH207	5m	16m	3m	1m	**53s**
AAH055	4h	8h	27m	**4m**	43m
AAH015	7h	6h	4m	**1m**	1h
MAE016	**20s**	-	1m	1m	3m
AAH020	8h	11h	2h	**23m**	3h
MAE008	**20s**	-	6h	27m	5m
AAH024	**15s**	15s	1m	16h	-
AAH045	49s	**15s**	15h	3h	-
MAE014	**20s**	-	23h	2h	2h
AAH032	5h	21h	1h	**28m**	-
MAE104	3m	**2m**	22h	13h	16h
AAH014	20h	5h	**21m**	21h	14h
AAH026	46s	**40s**	22h	22h	-
AAH007	**1m**	2m	22h	-	-
MAE115	**9h**	15h	-	19h	12h
AAH017	**7h**	-	21h	10h	20h
JCH201	**4h**	-	-	19h	21h
AAH001	**1h**	-	23h	-	-
AAH010	22h	-	**9h**	-	-

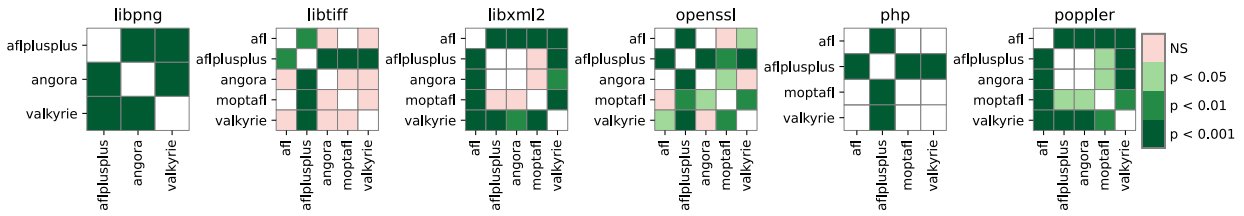


Figure 3.3: Siginificant plot of Valkyrie. Valkyrie is superior than state-of-the-art on libpng, libxml2, and poppler.

We list Valkyrie's performance on Magma in Figure 3.2. We calculate the arithmetic mean number of bugs found per trial per day. However, state-of-the-art fuzzers rely on randomized methods, a bug found in one trial may not be triggered in the another. Therefore, we also list all the unique bugs found, including bug id and the time used to trigger it in Table 3.5. The time shown is the arithmetic mean time to trigger a bug. If the fuzzer did not trigger a bug, then the time to trigger is set to 24 hours for that fuzzer. Therefore, for non-deterministic fuzzers, the mean time to trigger a bug becomes large when the bug is triggered only a few times. For example, AFL++ triggered the bug `AAH001` in a few minutes in only one trial, so the mean is 23h across 10 trials.

Valkyrie finds 21 unique integer and memory errors in Magma, while AFL, ALF++, MoptAFL, and Angora found 14, 19, 18, and 14 errors, respectively. Overall, Valkyrie ranked #1 and found 10.5% and 50% more errors compared with AFL++ and Angora, respectively. We conduct the Mann-Whitney U test to obtain p-value between each pair of fuzzers and list the siginificant plot of Valkyrie in Figure 3.3. Of 7 libraries, Valkyrie ranked #1 on libpng, libxml2, and poppler ($p < 0.001$ compared with #2); tied #1 on openssl and php ($p < 0.01$ compared with #3); tied #2 on libtiff. No fuzzer found any integer or memory errors on sqlite3. We want to emphasize that Valkyrie achieved the result with no randomization design.

Bug `AAH001` demonstrates that not only randomness is not required in certain bugs, but also that compensated steps can be effective in predicate solving. `AAH001` is a divide-by-zero in libpng. We listed the code snippet of `AAH001` in Listing 3.2. To trigger it the mutator must change `png_ptr->width` to `0x5555_5555` and `png_ptr->channels` to `3`, and the later two conditions to false. [52] proved that

```
1  // AAH001
2  size_t row_factor_l = 1 + (png_ptr->interlaced? 6: 0)
3   + (size_t)png_ptr->width
4   * (size_t)png_ptr->channels
5   * (png_ptr->bit_depth > 8? 2: 1);
6  size_t row_factor = (png_uint_32)row_factor_l;
7  if (png_ptr->height > PNG_UINT_32_MAX/row_factor) {...}
8  // MAE014
9  char *dir_start = value_ptr + maker_note->offset;
10 int NumDirEntries = php_ifd_get16u(dir_start, ImageInfo->motorola_intel);
```

Listing 3.2: Two seemingly easy bugs AAH001 and MAE014 in Magma. Valkyrie can trigger this bug in seconds while other fuzzers can take hours.

it is hard for the randomized method to trigger it and claimed that only a fuzzer with a solver could trigger this easily. However, Angora failed to trigger it. When Angora mutates the value close to `0x5555_5555`, even a small step in `png_ptr->channels` will overshoot and overflow the result. For example, when setting `png_ptr->channels` to `4`, the result will be a small value due to overflow; when setting to `2`, the result will be a large value. Angora may conclude that this variable has negative gradient and start moving it to a smaller value. When it happens, Angora may get the wrong gradient and cannot progress correctly. However, Valkyrie knows the upper bound of the unsigned value and forces the solver not to exceed it using compensated steps. Thus Valkyrie is able to solve it and triggered this bug within the hour in all ten trials.

Valkyrie is also the first to find many of the bugs compared with its peers. The reason is that solver-based fuzzers work on predicates in a more orderly manner. Randomized methods can be choked by a predicate, not knowing if it is a hard one or just infeasible, wasting its time budget. However, Valkyrie can report with confidence wether the predicate can be solved and explore a new path or report it unsolvable. One example is `MAE014` as shown in Listing 3.2. `dir_start` is a pointer to the buffer and `php_ifd_get16u` tries to get a `u16` from the buffer. However, it does not check whether `dir_start` is pointing to the last byte of the buffer, causing the code to over-read one byte from the buffer. In the case of Valkyrie, it will try to increase the index of the read by setting up a predicate $\texttt{dir\_start} + 1 > \text{MAX}$, thus triggering the bug in 20 seconds. However, it generally takes fuzzers in AFL family hours to trigger it. Furthermore, these two examples demonstrate that our design in

Section 3.3.3 is effective.

Valkyrie found four unique errors on libtiff (`AAH010`, `AAH014`, `AAH015`, and `AAH020`), the same number as other state-of-the-art fuzzers. However, on average, only three errors are triggered per trial because 24 hours timeout is not enough for Valkyrie. The seeds corresponding to `AAH010` and `AAH014` are scheduled with the same priority. There is no guarantee which one is taken out first. In any trial, if one seed was taken, the other would not be taken before timeout. Thus the mean time to trigger these two bugs are both 20+ hours.

We want to comment on another interesting finding regarding MoptAFL and AFL++. MoptAFL is reported to be the best fuzzer in this benchmark, however, in our experiment, MoptAFL found fewer bugs than AFL++. We carefully compared [52]'s result with ours and find that, in our experiment, AFL++ found several bugs that were reported as untriggered. Some examples include `AAH001, AAH007` in libpng, both of which are only triggered once by AFL++ across ten trials. The difference is surprising considering we used the same configuration provided by [52]. This further proves that randomized methods are volatile and unstable, while our deterministic approach is simpler and more reliable.

In summary, Valkyrie found 21 unique integer and memory errors on Magma, the most compared with other state-of-the-art fuzzers. Also, Valkyrie had little to no variance across ten trials, while others showed unstable performance. Therefore, we can answer **RQ1** with confidence that Valkyrie is state-of-the-art on Magma.

3.4.2 Real-world open-source programs

While performing well on Magma is sufficient to claim Valkyrie is state-of-the-art, we would like to evaluate on real-world programs and see the branch coverage data. Therefore, to demonstrate Valkyrie's effectiveness on real-world programs already in production, we selected a series of open-source programs to evaluate Valkyrie and demonstrate the effectiveness of its methods and techniques in real-world situations. Of these open-source programs, there are image processors (*jhead*, *imginfo*), binary file processing programs (*nm*, *objdump*, *size*, *readelf*), structured text parsing utilities

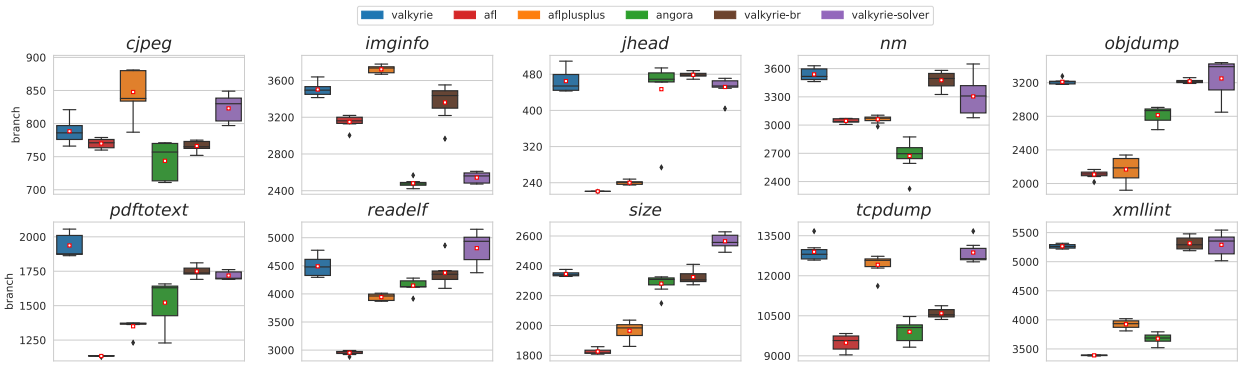


Figure 3.4: Branch coverage of six fuzzers in 24 hours time. Valkyrie-br is Valkyrie with only branch coverage improvement, Valkyrie-solver is Valkyrie with only solver improvement. Both design increased branch coverage compared with Angora in all programs. Overall, Valkyrie ranked #1 on geometric mean number of branches reached.

(*xmllint*), pdf parsers(*pdftotext*), network utilities(*tcpdump*). Because different tools count branches differently, for fairness of comparison, all branch coverage reported are generated by afl-cov [108].

The results of these experiments are shown in Figure 3.4. We obtain p-value between each pair of fuzzers using Mann-Whitney U test. Valkyrie ranked #1 on seven out of ten applications ($p < 0.01$ compared with #2), #1 tied with Angora ($p = 0.0011$ compared with #3) on *jhead*, #2 on *cjpeg* and *imginfo* ($p < 0.05$ compared with #3).

We also plotted the branch coverage against time in Figure 3.5. Valkyrie is the fastest fuzzer in all programs except *imginfo*, i.e., Valkyrie spends less time to reach the same branch coverage compared with other fuzzers. This trend is clearest in *objdump*, *readelf*, and *size*. It further demonstrated the effectiveness of deterministic algorithms we introduced in branch counting and solver. While state-of-the-art fuzzers are mutating randomly without knowing the detail of the program, Valkyrie can flip a predicate within several steps.

We also wish to demonstrate that a more sophisticated solver algorithm is a suitable trade-off between execution throughput and mutation effectiveness. Thus we present the growth of branch coverage of Valkyrie as well as the three comparison fuzzers AFL, AFL++ and Angora over its 24-hour-run on programs *objdump* and *pdftotext* in Figure 3.6. In the case of *objdump*, the growth of Valkyrie's coverage is slower than that of Angora's, but Valkyrie's solver can solve a greater

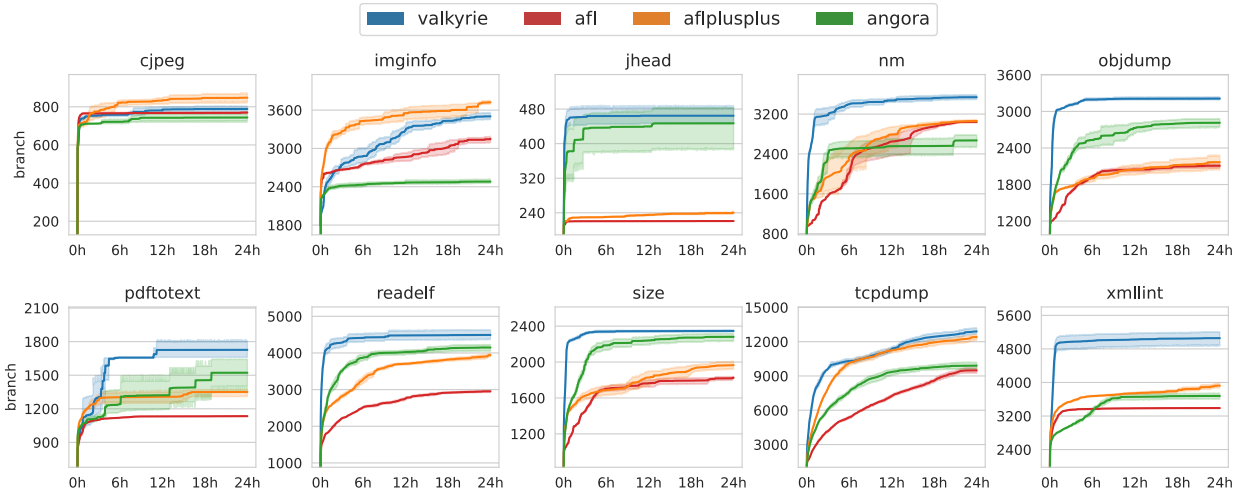


Figure 3.5: Branch coverage of four fuzzers in 24 hours time. Valkyrie not only finds more branch coverage but also is the fastest one on eight of ten applications thanks to deterministic algorithms.

Figure 3.6: Plots of branch coverage over 24 hours for all evaluated fuzzers on *objdump* and *pdftotext*.

number of predicates, resulting in Valkyrie's lead over Angora and the other fuzzers after around 10 hours. In the case of *pdftotext*, both Angora's and Valkyrie's growth is slower than that of AFL++'s. However, both can solve a *critical branch* that increases their coverage significantly, with Valkyrie being the faster of the duo. Besides, Valkyrie is capable of solving more branches than Angora over 24 hours.

In summary, the geometric mean number of branches Valkyrie reached per target is 2452, 8.2% and 12.4% more than AFL++ (2266) and Angora (2181), respectively. We can answer **RQ2** with confidence that Valkyrie is the state-of-the-art on real-world open-source programs.

3.4.3 Effectiveness of deterministic branch counting

We wish to understand the advantages of Valkyrie's branch counting mechanism quantitatively. We first controlled the variable to see how much improvement collision-free context-sensitive branch counting design contributes. Therefore, we disabled our improved solver and compared it with Valkyrie and Angora. The result is shown in Figure 3.4, the modified version is labeled as Valkyrie-

br. We find that Valkyrie-br outperformed Angora in all cases, proving that this design is effective. Our study shows the improvement is contributed by two designs: branch instrument optimization and context-sensitive collision-free branch counting.

We first examined the effectiveness of our branch table optimization strategies by obtaining the buffer sizes required by Valkyrie, as shown in Column 2-4 in Table 3.6. We observe that our optimization strategies can reduce the bitmap size by 69% on average. We used seeds generated by AFL++ to evaluate how much runtime is reduced. Column 5-7 in Table 3.6 show that we reduced runtime by 28% on average. Thus, given the same amount of time, Valkyrie can test the program more.

We then analyzed the buffer utilization rates of AFL and Angora under the evaluated programs. By default, AFL uses a 64K buffer. Angora uses 1M to allow context-sensitivity. The utilization rate is shown in Columns 2 and 4 in Table 3.7. Many programs' utilization rates exceed the recommended limit of 4%, even ranging up to nearly 34%, indicating that a newly found branch has a nearly 34% chance of colliding with existing branches. Under the default settings, many instances have a high potential for branch collisions, as evidenced by the high bitmap utilization rate of up to around 36%. Therefore, the default buffer sizes are too small for ordinary programs.

We further resized their bitmaps according to the size required by Valkyrie to achieve collision-free branch counting and analyzed their utilization rates. Their bitmap sizes should be a strict power of 2, so we found the closest value possible for each program, as listed in Column 7 in Table 3.7. We list the utilization rate under such sizes Column 3 and 5 in Table 3.7. The utilization rates have dropped to under 4% since we increased AFL's buffer size for most programs. However, AFL lacks context-sensitivity and can potentially lose the capability to identify branches that increase the overall coverage. Angora, on the other hand, still exceeds the recommended limit greatly in many cases, resulting in significant accuracy loss. In comparison, Valkyrie guarantees accuracy while maintaining context-sensitivity, which is the second reason why the branch coverage increased in Figure 3.4.

Therefore, we can answer **RQ3** with confidence that Valkyrie's branch counting mechanism is a

Table 3.6: Bitmap size for Valkyrie before and after optimization. On average we reduced 69% of all instrumentations and 28% of runtime.

Program	Valkyrie bitmap size (B)			Valkyrie bitmap runtime (µs)		
	Original	Optimized	Reduction	Original	Optimized	Reduction
cjpeg	254,874	74,576	70.74%	10,331	7918	23.35%
imginfo	133,010	34,690	73.92%	20,769	12,583	39.41%
jhead	13,620	4396	67.72%	1124	776	30.92%
nm	1,758,594	542,688	69.14%	1491	1270	14.84%
objdump	2,196,528	691,048	68.54%	1405	1374	2.24%
pdftotext	400,858	112,808	71.86%	6312	5663	10.29%
readelf	353,222	132,352	62.53%	1229	902	26.57%
size	1,750,206	540,180	69.14%	1687	1359	19.44%
tcpdump	1,554,400	506,468	67.42%	1278	972	23.93%
xmllint	3,323,032	996,220	70.02%	1439	1115	22.52%
Total	11,738,344	3,635,426	69.03%	47,065	33,932	27.90%

better trade-off and outperforms that of comparable fuzzers.

3.4.4 Effectiveness of deterministic solver

In Figure 3.4 we evaluated Valkyrie with only solver enabled, the modified version is tagged as Valkyrie-solver. The result shows that we improved branch coverage compared with Angora in all open-source programs. While it is enough to evaluate the triggered bugs and branch coverage, we would like to evaluate how compensated step helps the solver. Since both Valkyrie and Angora are fuzzers based on gradient descent, we test the solver's performance separately.

Although Valkyrie-solver is better than Angora in Figure 3.4, it would be unfair if we directly compare the number of predicates solved in that experiment for the following reasons: 1) The initial state of each solver is not the same. Listing 3.1 has shown that given a bad initial point, even a fairly "easy" predicate can be hard to solve. 2) Different sets of predicates may be presented to two solvers. When a unique predicate is solved by only one solver, the predicate may reveal a new path with predicates that are not known to the other solver. 3) Different time budgets and scheduling algorithms. Angora devotes much of its time to random search, which means the solver may not

Table 3.7: Bitmap utilization for AFL and Angora on open-source programs. We evaluated their respective utilizations under default sizes and adjusted sizes. "*" indicates failure, AFL refuses to run *jhead* with only 8K bitmap.

Program	AFL utilization		Angora utilization		Bitmap size (B)	
	Default (64K)	Adjusted	Default (1.0M)	Adjusted	Valkyrie	Adjusted
cjpeg	2.11%	1.06%	0.24%	1.88%	74K	128K
imginfo	10.23%	10.30%	1.68%	23.94%	34K	64K
jhead	0.45%	*	0.54%	49.51%	4.2K	8.0K
nm	7.92%	0.49%	33.14%	33.14%	542K	1.0M
objdump	5.26%	0.33%	24.98%	24.96%	691K	1.0M
pdftotext	3.30%	0.83%	18.88%	56.67%	112K	256K
readelf	10.92%	2.73%	4.05%	15.24%	132K	256K
size	4.49%	0.28%	14.75%	14.72%	540K	1.0M
tcpdump	20.85%	2.59%	34.64%	57.13%	506K	512K
xmllint	6.51%	0.41%	18.30%	18.29%	996K	1.0M

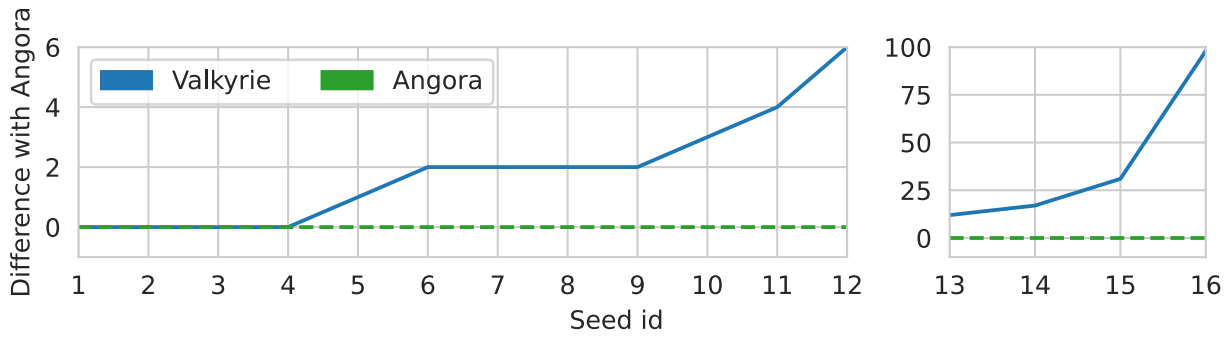


Figure 3.7: The difference between preidcates solved by Valkyrie and Angora over 16 seeds.

have enough time to solve the predicates.

Therefore, to provide a fair comparison environment, we randomly selected seeds generated by AFL++. For each seed, we run both fuzzer three times with that seed being the only initial seed, solving problem 1). We also disable fuzzers branch counting ability, so new seed is not added to the seed queue, solving problem 2). There is no timeout in each experiment, the fuzzer will stop when all predicates are either solved or discarded. This solves problem 3).

The predicates presented in the initial seed and the number of solved predicates by both solver are plotted in Figure 3.7. We can find that given the same setting, Valkyrie is able to solve more

Table 3.8: Bugs found by Valkyrie and Angora. Valkyrie found six bugs in three programs while Angora only found three.

Program	Description	Bugs found by each fuzzer			
		Angora	Valkyrie	Valkyrie-br	Valkyrie-solver
cjpeg	Floating point exception	✔	✔	✔	✔
imginfo	Assertion failure-1 (`qmfbid == JPC_COX_RF`)	✔	✔	✔	✔
imginfo	Assertion failure-2 (`absstepsize >= 0`)		✔	✔	✔
imginfo	Assertion failure-3 (Check Section 3.4.5 details.)		✔		✔
pdftotext	Throwing `GMemException`	✔	✔		✔
pdftotext	Out of bound read		✔	✔	

predicates than Angora on 12 out of 16 seeds. Four seeds (seed 13 to seed 16) saw a more than ten predicate increase. Seed 16 solved 92 more predicates than Angora. This is because these four seed are taken from *tcpdump* and *xmllint*'s queue, where more predicates are presented in a seed. Therefore, we do not consider them as outliers in our data.

On average, Valkyrie can solve 11 more predicates per seed. Notice that this is the solver's improvement over *one* seed, when in real-world fuzzing senario, thousands of seeds will be generated over the course of 24 hours. This gives us a positive answer to **RQ4**, the compensated step does improve the solver performance.

3.4.5 Bug finding ability of Valkyrie

In Section 3.4.1 we find that Valkyrie can find most memory and divide by zero errors compared to the state-of-the-art. Although examples like `AAH001` have demostrated solver's effectiveness, we wish to carry out a more detailed study to understand each components individual contribution to bug finding in real-world settings. We first instrument programs in Unifuzz [105] using the approach described in Section 3.3.3, then compiled these programs using Angora, Valkyrie, Valkyrie-br, and Valkyrie-solver. Similar to previous evaluations, we run ecah fuzzer for 24 hours and ten times. After fuzzing we collect *all* errors the fuzzers found, deduplicate them and found the following bugs in three programs. The detailed bugs are listed in Table 3.8.

We find that Valkyrie is able to find six bugs while Angora only found three. Valkyrie-br

```
1  int Catalog::countPageTree(Object *pagesObj) {
2    for (i = 0; i < kids.arrayGetLength(); ++i) {
3      kids.arrayGet(i, &kid);    // Access without check
4      n2 = countPageTree(&kid);
5      if (n2 < INT_MAX - n) {
6        n += n2;
7      } else {
8        error(errSyntaxError, -1, ...);
9        n = INT_MAX;
10     }
11     kid.free();
12   }
13 }
```

Listing 3.3: Code snippet copied from xpdf. The program accesses the array without checking the bound.

and Valkyrie-solver found four and five bugs respectively. The main difference comes from three assertion failures in *imginfo*. The first and second assertion failure are `qmfbid == JPC_COX_RF` and `absstepsize >= 0`, which can be solved by magic byte matching quickly, thus all fuzzers triggered it. However, the third assertion failure is `!((expn + (numrlvls - 1) - (numrlvls - 1 - ((bandno > 0) ? ((bandno + 2) / 3) : (0)))) & (~0x1f))`, which involves three variables and a nested condition. Such predicate requires that the last five bits of the result are not all zeros. Tradition solvers like Angora's strugle to calculate the gradient when it reaches the boundaries, thus unable to solve it. On the other hand, only Valkyrie and Valkyrie-br triggered an out of bound read. After some study we find that the program attempt to access a buffer without checking the index(Listing 3.3). We find that using Angora's branch counting, the loop back edge colided with other edge. When the other edge is executed repeatedly, Angora had no motivation to increase the iteration of the loop, since to Angora's eyes this has already been executed. Since there is no collision in our branch counting, Valkyrie will try to increase the numer of iteration until its greater than 128, a heuristics number set by [1, 53].

In summary, Valkyrie found six unique errors in three programs, ranked number one compared to Angora and other varients of Valkyrie. We can answer **RQ5** with confidence that both deterministic branch counting and solver contributed to Valkyrie's bug finding capability.

3.4.6 Summary

In the previous sections, we have addressed all research questions. Our results show that Valkyrie triggers 21 unique integer and memory errors, 10.5% and 50% more than AFL++ and Angora, respectively. In real-world programs, Valkyrie reached 2431 branches per target on average, 8.2% and 12.4% more compared with AFL++ and Angora, respectively. We demonstrated that our branch counting mechanism is a better solution for efficient and accurate feedback. Finally, we demonstrated that our predicate solving algorithms works effectively on real-world branch predicates, allowing Valkyrie to perform better than the other fuzzers we use for evaluation. Thus we claim that Valkyrie, which utilizes accurate and efficient feedback and effective predicate solving, is principled and reliable.

3.5 Discussion

3.5.1 Unsolved predicates

Previous sections demonstrated the effectiveness of our solver. However, there are scenarios where our solver experience difficulty when solving branch predicates.

The solver is designed to solve single predicates. One possible cause would be unsolvable predicates guarding dead code, such as redundant error checks. The solver would also have difficulty solving some non-convex predicates, i.e., its local minima are not the global minimum.

If the predicate is nested, the solver may mutate the input and modify the outcome of its parent predicate(s), rendering the target predicate itself unreachable. We can solve this by applying Matryoshka's framework for solving nested branch predicate(s) [85].

On the other hand, Valkyrie relies on DFSan [80], which can be slow when input space is large. While deterministic methods are more predicable and stable, the extra workload may prevent it from scale to large applications. Therefore, we would suggest a combined approach where we can use Valkyrie to explore hard to trigger predicates, and use AFL++ to explore the code.

3.5.2 Bug detection

We may have missed bugs in Magma due to the following reasons. Apart from the aforementioned issues, one main reason is that our work focus on increasing program coverage, our exploitation instrumentation only targets a small subset of bugs. Many common problems such as null pointer dereference, double free, use after free, etc. are not in the scope of this chapter. However, we managed to find more bugs in three libraries in Magma and improved branch coverage by 12.4% compared to Angora. This fact further proved that Valkyrie is a reliable tool.

3.5.3 Branch counting effectiveness

In Figure 3.4, we find that Valkyrie-solver can reach more branches than Valkyrie in rare cases, e.g. *readelf*. Although the only difference between Valkyrie-solver and Valkyrie is the our branch counting method, this does not suggest our method is less effective. Valkyrie-solver performed better because branch counting with branch collisions may miss many branches. These missed branches have two-sided effects. On the one hand, there may be key branches that lead to more coverage, thus limiting solver's ability. On the other hand, some difficult conditions are not generated in the first place, thus saving fuzzer's time. When the former effect is in dominance, Valkyrie will outperform Valkyrie-solver, vice versa. These two-sided effects are neither predictable nor desirable, which further justifies our motivation to eliminate branch collisions.

3.6 Related work

3.6.1 Branch counting methods

Since AFL, much work has been devoted to strike a balance between branch counting sensitiveness and the probability of colliding. Angora [44] updated AFL's method by adding a function context to the branch counting table. CollAFL [31] proposed replacing AFL's random ID generation with one that would largely prevent duplicate edge IDs from occurring. However, its method is subject to the bitmap size exceeding the number of branches in the target program and cannot integrate

context-sensitivity easily like Valkyrie did. Recent work [35] points out that branch counting is a trade off. Fuzzers benefit from sensitive branch counting algorithms, yet the more sensitive it is, the more computing budget is consumes. Angora uses an enlarged branch counting table to allow context sensitivity. However, that brings substantial memory overhead to the fuzzing process. This problem is sloved by Valkyrie by making branch counting collision free and reducing the size of matcher table.

3.6.2 Predicate Solving Methods

Many work focuses on predicate solving. Solver-based fuzzers aim at better solvers on branch predicates to reach high code coverage. REDQUEEN [43] solves hashes and checksums through input-to-state-correspondance. KLEE [109] uses symbolic execution to solve predicates in the program to generate seeds, but symbolic execution can be ineffective when program path is deep and nested. This made KLEE ineffective compared to Valkyrie. Angora [44] solves branch predicates using principled methods such as gradient descent, yet in this chapter we show that without continuous assumption Angora's solver may fail some simple cases. Angora's method is largely based on mathematical optimization through gradient descent. However, Angora cannot effectively solve branches that are nested together. Matryoshka [85] proposes procedural methods for solving nested constraints in real-world situations. Matryoshka solves nesting branches by a slightly modified gradient solver, which is ad hoc and unjustifiable.

3.6.3 Targeted fuzzers

Targeted fuzzers attempt to target the potentially buggy code. AFLGo [77] proposed to reach the target code by giving priority to those close to the target. IJON [36] tries to manually take out some "important" program points and ask the fuzzer to put more time on it. However, unlike Valkyrie which can identify potential buggy code automatically, both AFLGo and IJON requires a human expert to label the target. IntEgrity [107] specifically target on integer errors and automatically identifies them using static analysis. Unlike Valkyrie who targets integer errors and memory errors,

IntEgrity only targets integer errors, limiting its scope. Savior [110] also targets potential bugs, but it focuses on using seed scheduling to find those that lead to potential buggy code. TOFO [111] proposed a method to calculate the distances between all basic blocks in seed and target basic block and reaches its target by always selecting the closest seed. Both Savior and TOFO focuses on scheduling to reach targeted code faster and neglected exploration part of the fuzzing. Therefore, their tool is useful when reaching target for exploitation is more important than exploration. Besides, all the tools use randomized approach to reach a target, while only Valkyrie deals this by using a new solving method.

3.6.4 Machine learning based fuzzers

Machine learning has become more and more popular in various areas. There have also been many attempts to incorporate it into fuzzers. [27] attempts to use a neural network to smooth the predicate to predicate a gradient when it can't be calculated. [112] uses data driven method to learn the input format from valid seeds to generate other seeds. [113] uses generative model to generate C programs to fuzz compilers. However, their work only tests C compiler, which is a very narrow scope. Large language models (LLM) have also started to play a role in the fuzzing community. [114–117] seek to use LLM to generate or mutate inputs for fuzzing. On the other hand, [118] attempts to reach more code by using an LLM to select program arguments. However, we believe machine learning are black boxes that can't be reasoned. While these approaches trump in fuzzing, a more deterministic base line is needed to provide a reasonable baseline.

Chapter 4

IRFuzzer: specialized fuzzing for LLVM backend code generation

4.1 Introduction

Modern compilers, such as LLVM [59], are complex software. For example, LLVM consists of over seven million lines of C/C++ code contributed by more than 2500 developers[1]. Given the size of this codebase and its importance in the computing ecosystem, an effective and scalable verification method is critical. Despite extensive regression testing and wide usage, latent bugs remain and their impact on users can be quite significant given the widespread distribution and long lifetimes of compilers.

To reduce latent bugs, various techniques have been used to automate the verification of compilers, such as partial model checking [119], fuzzing [60, 120, 121], and differential testing [122, 123]. Although end-to-end formal verification of compilers has been applied [124, 125], these techniques have not yet scaled to practical compilers such as LLVM, which support a wide range of architectures, programming languages, and use models, including just-in-time compilation and link-time optimization.

[1] `https://github.com/llvm/llvm-project`

In the specific case of LLVM, another factor making verification difficult is that the interface between the compiler optimization and machine code generation is widely used but not completely specified. As a result, it can be difficult for backend developers to understand whether they have completely implemented the wide range of possible inputs. In addition, backends often differ greatly in their relative code maturity, including some targets that are relatively mature and other targets for new devices that are in active development.

We find that the state-of-the-art fuzzers failed to find new bugs of a compiler backend for various reasons. General-purpose fuzzing techniques, such as AFL++ [53], often do not consider input validity and struggle to explore control paths in the compiler backend since most binary strings are invalid compiler inputs. In order to test the compiler backend more effectively, we aim to generate LLVM Intermediate Representation (LLVM IR) that complies with the language reference. LLVM includes `llvm-opt-fuzzer` and `llvm-isel-fuzzer` that generates valid IR for middle end and backend fuzzing, respectively [126]. Both of them are based on the library FuzzMutate [127] for valid IR mutation. However, FuzzMutate can't construct complex control flow, and it only generates a few instructions with scalar types. On the other hand, end-to-end fuzzing tools, such as CSmith [60] and GrayC [3], test the whole pipeline of the compiler, but they cannot to efficiently explore control paths in the compiler backend. CSmith does not take any feedback from the compiler, which contributes to its ineffectiveness. A more fundamental reason is that front-end parser and middle-end optimizations may limit the set of features seen by the compiler backend. High level languages like C may not exercise all backend features in LLVM. Therefore, even if GrayC used branch coverage feedback from libFuzzer [86], it missed many backend bugs introduced before LLVM 12, which were found by us. As a result, when a new language, such as Rust, is introduced, new backend bugs may still arise [128].

Generating valid IR is challenging with three major difficulties. In order to generate a complex control flow graph (CFG), we have to maintain all data dependencies to avoid use-before-definition situations. A valid CFG can be easily invalidated by a jump, as shown in Figure 4.2. This challenge does not exist in C generation if one does not generate `goto` statements. Besides, modelling the

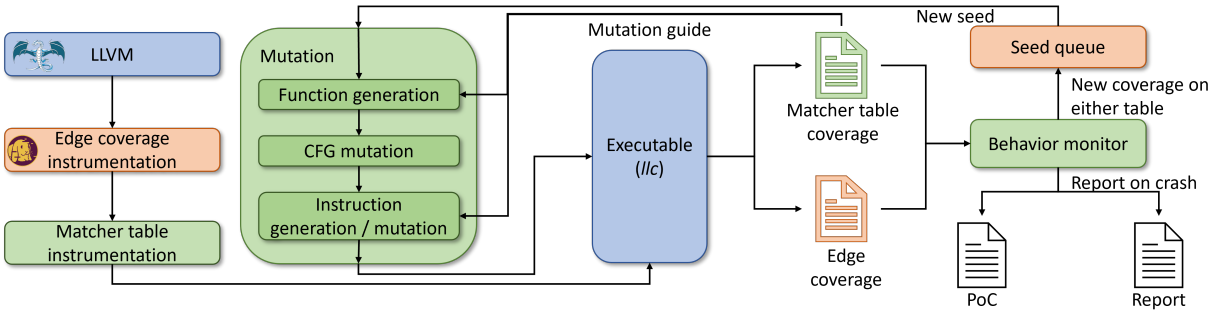
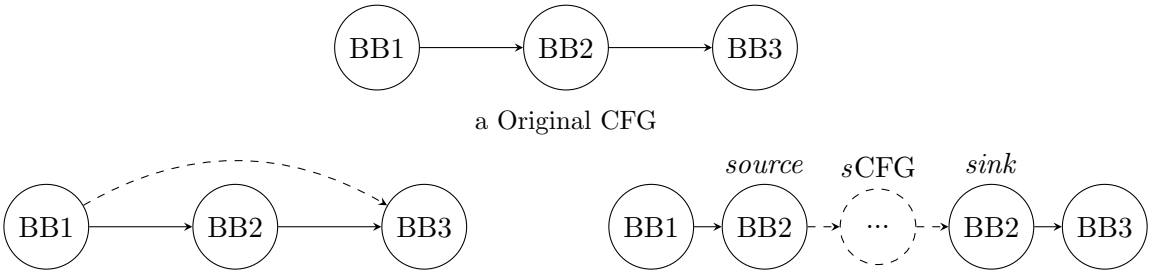


Figure 4.1: Overview of IRFuzzer. Green shaded components are the contributions of this chapter, orange shaded components are AFL++, and blue shaded components are from LLVM. We first create a LLVM IR mutator that guarantees the correctness of the generated input (Section 4.3.1). We introduce a new coverage metric to keep track of the backend code generation while providing a mutation guide to the mutation module (Section 4.3.2).

instructions missing in FuzzMutate isn't trivial. We must make sure that the types of the operands in each IR instruction match, but enumerating the large numbers of natively supported vector types is infeasible. Finally, it is difficult to model intrinsic functions for all architectures, as intrinsics are often poorly documented and vary from architecture to architecture.

We also observe that AFL++'s feedback mechanism performed poorly when testing the backend. It uses branch coverage as feedback, which runs into severe branch collision problems when fuzzing large codebase such as LLVM. Naively increasing the branch counting table size introduces huge overhead [31]. A more fundamental reason is that much code generation logic in the LLVM backend is implemented using table-driven state machines. A matcher table encapsulates all possible states as a constant byte array, meaning that branch counting can't observe this logic during fuzzing. The fuzzer needs a better feedback on whether the seed is interesting or not. If the seed is not interesting, the feedback should also inform the mutator what type of input is more desired.

To address these issues, we design a specialized fuzzer, IRFuzzer, for fuzzing the LLVM compiler backend. Figure 4.1 shows the overall structure of IRFuzzer. We first design a mutator that generates valid IR (Section 4.3.1). We maintain the correctness of CFG during mutation. We also use a descriptive language to list the requirements of each instruction type. This approach ensures that inputs to the compiler backend are always valid, increasing the efficiency of fuzzing. Our work



a Original CFG

b Naive mutation by random edge insertion. After mutation BB2 no longer dominates BB3, thus every value in BB3 referencing BB2 will invalidate the whole module.

c Mutation by *s*CFG insertion. It breaks BB2 into two halves. No matter how complicated *s*CFG is, the domination relations are unchanged.

Figure 4.2: Examples of failed and successful CFG mutations, respectively.

expands the FuzzMutate to include aspects where compiler backends often have special handling, including multiple basic blocks with complex control flow, function calls, intrinsic functions, and vector types. Using IRFuzzer, we are able to generate a wider range of instructions and explore control paths in the compiler backends more efficiently.

Then, we introduce a new coverage metric (Section 4.3.2) by instrumenting the table-driven state machines in LLVM, enabling the design space to be more efficiently explored. New entries that are covered in the matcher table means new features are executed. Working together with branch coverage, they can provide a better feedback on whether a seed is interesting or not. Furthermore, the matcher table has all information about the instructions and intrinsics in one architecture. As a result, we use the matcher table to determine which instructions and intrinsics haven't been fuzzed. We design a feedback loop from the matcher table coverage to our mutator. IRFuzzer will periodically generate a coverage report containing the states that haven't been executed. The report will be sent to the mutator to guide future mutations, enabling IRFuzzer to test on different backends with no prior knowledge of the architecture.

We evaluate IRFuzzer on 29 mature backend architectures in LLVM (Section 4.5). Our results show that IRFuzzer is more effective than the state-of-the-art fuzzers such as AFL++ and GrayC. IRFuzzer generated inputs code with better edge coverage and matcher table coverage on 28 LLVM backends. Leveraging these techniques, we were able to find and report 78 confirmed, new bugs in

LLVM, 57 of which have been fixed, five have been back ported to LLVM 15. This demonstrates the high impact on improving the correctness of LLVM backend targets.

This chapter uses LLVM to demonstrate the importance of having a specialized fuzzer for the compiler backend. Since modern compilers have similar intermediate representations, we expect that our approach can be easily applied to other compilers without requiring heavy engineering efforts. We made the following contributions:

- We have designed and implemented IRFuzzer. To the best of our knowledge, IRFuzzer is the first backend fuzzer that uses coverage feedback to guide IR mutation.
- We compared IRFuzzer with other state-of-the-art fuzzers on LLVM upstream and found it to be the most effective on matcher table coverage metric.
- We carefully analyzed and categorized the bugs we found during our testing. In total, we discovered 78 confirmed new bugs in LLVM, 57 of them have already been fixed, five have been back ported to LLVM 15.

4.2 Background

4.2.1 LLVM

LLVM [59] is a mature compiler framework consisting of many components that can be targeted to different architectures. At its core lies the LLVM Intermediate Representation (LLVM IR), which serves as a target-independent abstraction separating the concerns of high-level programming languages from the low-level details of particular architectures. LLVM can be roughly partitioned into three layers as shown in Figure 4.3: A *frontend*, such as `clang`, translates programming languages to LLVM IR, including lexer, parser, AST transformation, etc. The *middle-end*, called `opt`, processes LLVM IR and performs common target-independent optimizations. The *backend*, called `llc`, converts LLVM IR to a target-specific machine code representation and eventually emits binary or assembly code for the target architecture. The LLVM backend supports multiple target

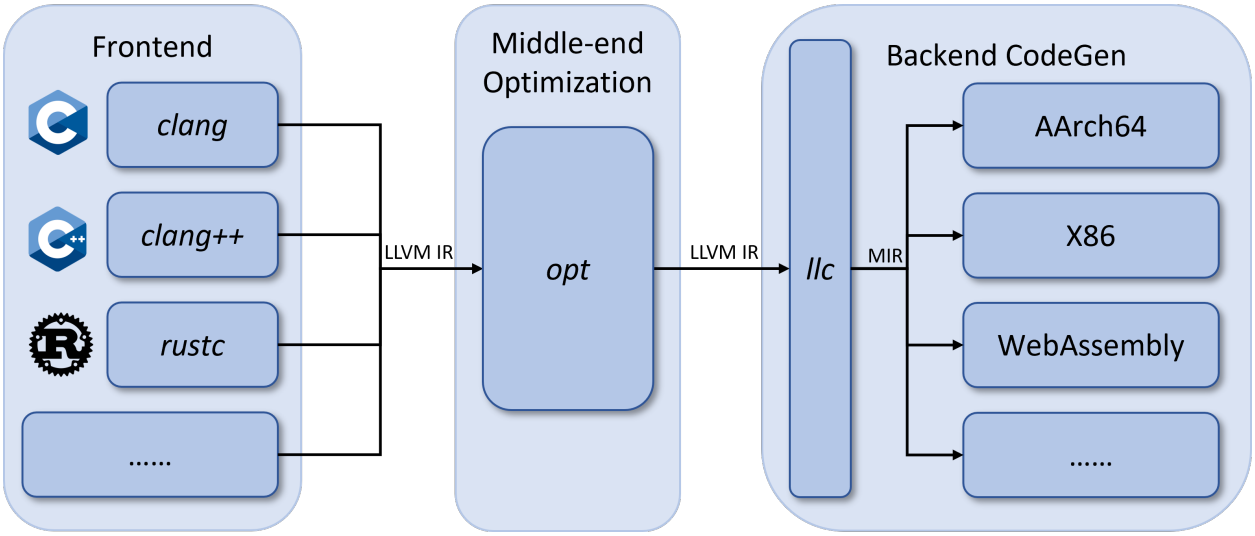


Figure 4.3: LLVM can be roughly partitioned into three components, frontend, middle end, and backend.

architectures through a plug-in abstraction, and the code to support a target architecture typically involves the implementation of API functions to describe common aspects along with target specific code to implement more unusual concepts.

The LLVM IR describes a static single-assignment (SSA) form [129], with a fixed set of instructions. Instructions are strongly typed, and the type of each value must match between its definition and all uses. A wide range of types are supported, including integers with arbitrary bit width up to 65,536 , floating point values, pointers, vectors, and other aggregate types. As with most high-level languages, LLVM IR allows the definition of functions, and the control flow between functions is implemented using the `call` instruction. Architecture specific intrinsic have no corresponding IR instructions, but are represented as function calls at IR level.

Control flow within a function in LLVM IR is represented using basic blocks and branch instructions. Special PHI instructions allow instructions in a basic block to refer to values defined in other basic blocks. Therefore, PHI instructions must respect control flow constraints and may only refer to values defined in predecessor blocks. This *domination constraint* [130] means that techniques used in high-level language generation cannot be easily adapted to LLVM IR.

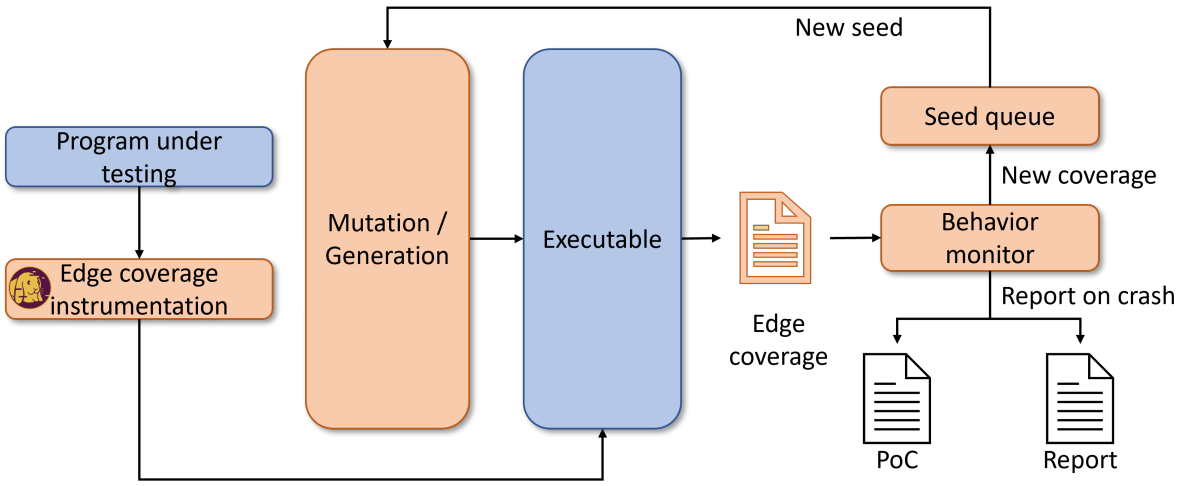


Figure 4.4: AFL can be modelled as a four-stage loop that tests the executable repeatedly.

The process of *instruction selection* in the LLVM backend replaces target-independent LLVM IR instructions with target-specific machine code instructions. LLVM provides two different frameworks to implement instruction selection that may be leveraged by the target backend plug-in. SelectionDAG [131] is the more mature instruction selection framework and is leveraged by all targets. In SelectionDAG, the code in each basic block is converted into a directed acyclic graph (DAG) representing the data dependency between instructions, and instruction selection is performed on the DAG. Since SelectionDAG processes each basic block independently, it can miss opportunities for optimization across basic blocks. GlobalIsel [132] is a newer framework that is only leveraged by some targets. GlobalIsel preserves the basic block structure within a function during instruction selection, enabling more optimization opportunities.

Both frameworks use *patterns* to describe rewrite rules applied during instruction selection. Some patterns are relatively simple and replace a single LLVM IR instruction with a single machine instruction. More complicated patterns may replace multiple LLVM IR instructions, or generate multiple machine instructions. Patterns may also include complex predicates to limit their applicability only to specific situations. For example, a pattern may only apply when a particular operand is a constant, or a certain hardware feature is enabled.

Most patterns are described declaratively in an LLVM-specific language called TableGen [133].

```
1  void SelectCodeCommon(SDNode *N, char *MatcherTable) {
2   bool Result;
3   while (true) {
4    switch (MatcherTable[Idx++]){
5      case OPC_CheckOpcode: {
6       uint16_t Opc = MatcherTable[Idx++];
7       Opc |= (unsigned short) MatcherTable[Idx++] << 8;
8       Result = (Opc == N->getOpcode());
9      }
10     case OPC_CheckPredicate:
11         ...
12     ...
13   }
14  }
15 }
```

Listing 4.1: SelectionDAG in LLVM that consumes a matcher table to do instruction selection.

In order to optimize the application of patterns, TableGen translates individual patterns into a state-machine representation implemented as a large byte array in C++ known as the *matcher table*. During compilation, the state machine in the matcher table is executed on each IR instruction and determines the correct pattern (if any) to apply. Listing 4.1 is a C++ code snippet used to evaluate the matcher table in SelectionDAG. `SDNode` is a data structure that represents an IR instruction. The while loop iteratively reads a command from the matcher table based on the current state, represented by the `idx` variable, evaluates the command, and selects the next state that will be evaluated. For example, `Opc_CheckOpcode` will check if the opcode of a given `SDNode` representing an instruction in the SelectionDAG graph matches a particular opcode. The `Result` will be used in future iterations, depending on the next entry in the matcher table. Evaluation of the matcher table continues until a single pattern is selected, or a state is reached where no patterns can apply.

Note that all patterns are evaluated using the same set of conditional branches in the switch statement in Listing 4.1. As a result, control flow coverage in the code is a **poor** indicator of whether all patterns have been exercised.

4.2.2 Coverage guided fuzzing

American Fuzzy Lop (AFL) [1] is a widely used open-source fuzzing framework that implements a form of coverage-guided fuzzing, Figure 4.4 shows an overview of AFL. Rather than simply generate

arbitrary inputs to a program under test (PUT), AFL instruments the PUT with the ability to track control-flow coverage. When a particular program input results in increased code coverage, AFL stores this input in a *seed cache* for future use. When generating new random program inputs, AFL prefers to select previous inputs from the seed cache and further mutate them, rather than generating completely random input. Using this strategy, AFL and coverage-guided fuzzing tools are able to more quickly explore all different control-flow paths of the PUT, when compared to black-box fuzzing techniques without instrumentation.

Many variations of coverage guided fuzzing have been developed, with the goal of finding bugs more efficiently by exploring a wider range of program behaviors with future executions of the PUT [134]. There are studies on the impact of different feedback algorithms [35, 36, 40]. Different methods are proposed to prioritize seeds to improve the performance of fuzzing. [38, 39, 42]. Some fuzzers also target on triggering specific bugs [15, 48, 107]. More advanced mutation strategies also show better fuzzing performance compared with random mutation [27, 30, 43–45]. Many improvements of have been implemented in AFL++ [53], making it a good framework for further development.

LLVM also introduces its own coverage guided fuzzing framework libFuzzer [86], coupled with FuzzMutate [127], it can be used to fuzz LLVM backend. However, FuzzMutate only generates a limited type of code and is not under active development. Still, the framework provides us with helpful insights into how should we mutate LLVM IR.

4.2.3 Challenges in compiler fuzzing

We believe that compilers represent a particularly challenging area to apply fuzzing, due to the size and complexity of the PUT involved. First, the input program has to be semantically meaningful. With program context, many structured fuzzing techniques [55, 135] based on context-free grammar cannot be directly applied. For example, generating an IR instruction depending on a value that hasn't been defined yet may cause the module verifier to abort. While high-level languages use notions like scope or lifetime to notate whether a value can be used, LLVM IR does not have that.

We can only reason the lifetimes of values in basic blocks by static analysis. For two blocks A and B, only when A *must* be executed before B, or A dominates B, can B directly reference values in A. However, when changing control flows, it is very easy to break that domination relation. For example, in Figure 4.2b, by adding an edge to the CFG we may invalidate the whole module and be rejected. We have to carefully maintain the CFG so that if a A dominates B, the relation remains the same after the control flow mutation.

In addition, we have to make sure the input has the correct syntax. LLVM IR is a strongly typed language with numerous types, including vector types and struct types, making it infeasible to explicitly enumerate types of legal operations. This challenge doesn't exist in some high-level programming language generation tasks like C [60, 121] and JavaScript [4, 55, 56, 136]. What's worse, each architecture can implement its customized LLVM IR instructions called intrinsic functions. The internal definition of intrinsic are often poorly documented, as the implementation details are often proprietary. These constraints make it hard to enumerate and model all of them without architecture specific knowledge.

LLVM IR implements an SSA representation of code, which only allows each variable to be assigned once. Consequently, it is very easy to reason if a variable is used at static time. If an IR instruction is not used, it is a dead code and the compiler erases it. Therefore, we wish our generated instructions to rely not only on constants but also other instructions.

Finally, machine instructions do not correlate with instruction selection's control flow, rendering traditional code coverage ineffective. When compiling LLVM IR to binary executable, both backend algorithms (SelectionDAG and GlobalIsel) use a table-driven method. Architecture developers will write code generation patterns in TableGen. These patterns will be compiled by LLVM into a static table known as a matcher table. The matcher table contains both data and control instructions for the pattern. At runtime, a while loop will consume this table. Thus, different instructions may be generated using the same control flow with different data.

4.3 Design

```
1  define i64 @f(i32 %I, <4 x i32> %V) {
2  Entry:
3      %ret_p = alloc i64, 1
4      %ret = load i64, ptr %ret_p
5      ret i64 %ret
6  }
```

Listing 4.2 A piece of LLVM IR program generated by function generation(Section 4.3.1.1). The function returns a 64 bit integer, so we allocate a stack memory and load from it to return. We will fill the memory in later mutations.

```
1  define i64 @f(i32 %I, <4 x i32> %V) {
2  EntrySrc:
3      %ret_p = alloc i64, 1
4      switch i32 %I, label %sCFG_Default [
5        i32 1, label %sCFG_1
6      ]
7  sCFG_Default:
8      br label %EntrySink
9  sCFG_1:
10     br label %EntrySink
11 EntrySink:
12     %ret = load i64, ptr %ret_p
13     ret i64 %ret
14 }
```

Listing 4.3 IR program mutated from Listing 4.2. Line Line 4 to Line 10 are introduced by *s*CFG insertion(Section 4.3.1.2). We insert *s*CFG by splitting the `Entry` block into two and generate a `switch` instruction.

```
1  define i64 @f(i32 %I, <4 x i32> %V) {
2  EntrySrc:
3      %ret_p = alloc i64, 1
4      switch i32 %I, label %sCFG_Default [
5        i32 1, label %sCFG_1
6      ]
7  sCFG_Default:
8      %I64 = zext i32 %I, i64
9      br label %EntrySink
10 sCFG_1:
11     %I1 = add i32 %I, 1
12     %J64 = call @f(i32 %I1, <4 x i32> %V)
13     br label %EntrySink
14 EntrySink:
15     %PHI = phi i64 [%J64, %sCFG_1], [%I64, %sCFG_Default]
16     store i64 %PHI, %ret_p
17     %ret = load i64, ptr %ret_p
18     ret i64 %ret
19 }
```

Listing 4.4 IR program mutated from Listing 4.3. Instruction insertion(Section 4.3.1.3) generated Line 8, Line 12, and Line 15. The placeholder memory is also used by `%PHI` to avoid undefined behavior (Line 16).

Figure 4.5: An example of how IRFuzzer mutates a module using different strategies.

To overcome those challenges in Section 4.2.3, we design IRFuzzer with two new components. Figure 4.1 shows the new components of IRFuzzer. During mutation, we first generate a function if there isn't one (Section 4.3.1.1). Then we change the control flow graph (CFG) to create more control flows (Section 4.3.1.2). Finally, we generate new IR instructions and mutate them (Section 4.3.1.3). Figure 4.8 shows an example of the mutation process using these mutation strategies. After mutation, we create a new method to measure the coverage of the program (Section 4.3.2).

4.3.1 LLVM IR mutation

In order to generate a wide variety of input while avoiding invalid inputs, we adopt a mutation-based strategy. This strategy starts with small valid seed inputs and modifies the seed inputs in ways that should also generate valid inputs. By randomly selecting between a number of small, well-defined mutations, we expect to eventually generate a broad class of valid inputs while avoiding invalid inputs. Figure 4.8 shows an overview of our design. We first generate an empty function is non is present (Listing 4.2). Then, we mutate the control flow by *s*CFG insertion (Listing 4.3). Finally, we modify or insert instructions in basic blocks (Listing 4.4).

4.3.1.1 Function generation

The LLVM backend has a significant amount of target-specific code related to function calls. As a result, it is important to generate a wide range of function definitions and function calls with different arguments and return types.

IRFuzzer implements a mutation strategy capable of generating new function definitions with arbitrary arguments and return types. One important constraint is that the return type of the function signature matches the type of each `return` instruction in the function definition. To ensure this, the function generation strategy also synthesizes a load instruction of an appropriate type as the operand for a `return` instruction. Although the value returned from the load may be uninitialized, later mutations may store values to the memory, validating the return value.

IRFuzzer also implements a mutation strategy to generate new `call` instructions which refer to

specific function declarations. The mutator is free to select any declared function and will generate compatible arguments and return values for the call, as with any other primitive instruction. Intrinsic functions are target specific operations that correspond to complicated machine instructions, and generating them will increase the code we can test. Yet they are treated as functions at middle-end. In particular, this mutation strategy will also select intrinsic functions to call.

4.3.1.2 Control flow graph mutation

Another area where target-specific code in the LLVM backend differs relates to control flow. Many machine code optimizations, such as jump threading, restructure control flow. In addition, certain compiler optimizations may select specific jump instructions, but this optimization can only be performed after instruction selection when code size and alignment are known. For instance, a common compiler optimization is to first select jump instructions into a "short" form with a limited offset range and then only later replace the short form with a "long" jump instruction if a larger offset is required. Control flow optimization can also affect register lifetimes, exercising target-specific code for spilling and restoring values from the stack.

IRFuzzer implements a structured approach to generating control flow. Inserting and removing arbitrary branches in the code can greatly change dominator constraints between basic blocks. For example, in Figure 4.2b, mutated from Figure 4.2a, BB2 no longer dominates $BB3$ after mutation. If any value in $BB3$ refers $BB2$, the module is invalid after mutation. We implement an elegant approach that uses sub-control flow graph, or sCFG, shown in Figure 4.2c. Instead of changing edges, we split a block and insert sCFG inside.

A sCFG is a CFG with a single *source* entry block and a single *sink* exit block that will be placed inside a larger CFG. Within the sCFG, we allow the synthesis of an arbitrary control flow graph. However, every control flow edge starting in the sCFG must be contained within the sCFG, except for *source* edges, *sink* edges, and return instructions. With this restriction, we can insert sCFG into a program without breaking the dominator constraint by randomly selecting a block and splitting it into two. The first part is *source* and the second *sink*. After block splitting, we generate

random *s*CFG starting from *source* and ending with *sink*.

The *s*CFG can be constructed with three main control schemas: *branch*, *switch*, and *return*, corresponding to different terminators of the basic block. We start with only one basic block and randomly select the terminator of the block. If the return schema is selected and the function requires a return type, we pick any value available that matches the return type of the function. If a branch or switch is selected as the terminator, we will find a previously generated non-constant value as a condition. If no such values can be found, we will allocate a stack memory as a placeholder. The branch can go to one of three places: *sink*, self-loop or return. If we generate a self-loop, we will also update all the PHI nodes in the block to include a new value.

Finally, all terminators will be generated when we are mutating CFG. Note that our instruction generation strategies will not mutate terminators in order to protect the integrity of the CFG.

4.3.1.3 Instruction modeling and generation

A key aspect of the LLVM backend is to convert the wide range of LLVM IR types to the (usually small) set of types natively implemented by each target architecture. Therefore, to exercise all features of code generation, it is necessary to generate IR instructions with as many data types as possible. However, many IR instructions only operate on a restricted set of data types, and FuzzMutate only modelled scalar types, which is trivial and limited. In order to model these restrictions for vector types, we categorize instructions as shown in Table 4.1. These definitions are reflected in the code as declarative declarations expressing both restrictions on the types of operands and constraints between the types of different operands. For example, the `anyIntOrVecInt` constraint restricts the valid types for a particular operand to be any integer type or vector of integer type. The `matchFirstOperand` constraint restricts the type of operand to be the same as the type of the first operand.

When generating a new instruction, we first randomly select an opcode and use the declarations to randomly select values that exist in the code with a compatible type. If no value exists with a compatible type, then the mutator will create a new operation with a compatible type. For

Table 4.1: Instruction modeling for IR instructions.

Operation type	Opcode		Argument descriptions		
Unary operation	fneg	:	anyFloatPointOrVectorFloatPoint		
Binary operations	add, sub, mul, (s|u)(div|rem)	:	anyIntOrVecInt	sameAsFirst	
	fadd, fsub, fmul, fdiv, frem	:	anyFPOrVecFP	sameAsFirst	
Bitwise operations	shl, lshr, ashr, and, or, xor	:	anyIntOrVecInt	sameAsFirst	
Vector operations	extractelement	:	anyVector	anyInt	
	insertelement	:	anyVector	matchScalarOfFirst	anyInt
	shufflevector	:	anyVector	matchLengthOfFirst	VecOfConstI32
Aggregate operations	extractvalue	:	anyAggregateOrArray	anyConstInt	
	insertvalue	:	anyAggregateOrArray	matchScalarOfFirst	anyConstInt
Memory operation	getelementptr	:	anySized	pointerOfFirst	anyInt
Casting operations	trunc	:	anyNonBoolIntOrVecInt	anyIntOrVecIntWithLowerPrecision	
	zext, sext	:	anyIntOrVecInt	anyIntOrVecIntWithHigherPrecision	
	fptrunc	:	anyNonHalfFPOrVecFP	andFPOrVecFPWHigherPrecision	
	fptoui, fptosi	:	anyFPOrVecFP	matchLengthOfFirstWithInt	
	uitofp, sitofp	:	anyIntOrVecInt	matchLengthOfFirstWithFP	
	ptrtoint	:	anyPtrOrVecPtr	matchLengthOfFirstWithInt	
	ptrtoint	:	anyIntOrVecInt	matchLengthOfFirstWithPtr	
	bitcast	:	anyType	anyTypeWithSameBitWidth	
Other operations	icmp	:	anyIntOrVecInt	sameAsFirst	
	fcmp	:	anyFPOrVecFP	sameAsFirst	
	select	:	anyBoolOrVecBool	matchLengthOfFirst	sameAsSecond

numerical types, the new operation could generate a random constant, undef, or poison.

In addition, a small number of operations are not modeled declaratively. For instance, `store` and `load` memory operations are structured differently enough from other operations that modeling them is not necessary. Some other instructions have constraints which are too complex to be simply handled in the declarative framework, and we resort to custom generators. For instance, instructions representing PHI nodes must be created with a number of operands equal to the number of predecessor blocks and must occur at the start of their basic block. Similarly, *call* instructions are handled manually too, since we must select a function declaration and find values that exactly match the operand types of the declaration.

To ensure that values are defined before they are used, the mutator searches for values defined in the following locations: global variable, function argument, values in dominators, and values defined by previous instructions in the same basic block. If no value with a compatible type exists, the mutator will attempt to generate a load from a compatible pointer, if one exists. Lastly, if a value with a compatible pointer type exists, the mutator will fall back to either creating a new global

variable, a new constant value, or a load from a stack memory location.

In some cases, the mutator may create IR instructions that define values which are never used. Since such dead code is likely to be removed by the compiler before instruction selection, the mutator will attempt to create a use for such values. One possibility is to store dead values to stack memory or a global variable. Alternatively, if there are instructions after the definition, or the current block dominates other blocks, the mutator may select an instruction with a compatible operand to replace.

When generating instructions, it is possible that the mutator allocated new stack memory as placeholders. In order to avoid undefined behavior, the mutator will again attempt to replace loads from these placeholders with other values of compatible type. If no such value exists, then the mutator will store a value into the placeholder location.

We don't model intrinsic functions, as they vary from architecture to architecture, potentially consuming a lot of time with little outcome. Instead, we rely on the feedback from matcher table coverage (Section 4.3.2.2). Matcher table coverage report contains a list of intrinsics that haven't been generated in the form of function definitions. Then, the mutator will randomly generate *call* instructions to intrinsic from this report.

4.3.1.4 Instruction shuffling

Another strategy shuffles instructions in the basic block. Changing instruction orders inside a basic block will change how backend does instruction scheduling. When shuffling, instruction orders need to be carefully handled otherwise a use-after-definition condition may arise. Our goal is to make sure that for each define-use edge, after shuffling define appears before use. This can be done by topological sorting.

We first remove all instructions except for the terminator from a basic block. Then we construct a directed acyclic graph from the dependencies between removed instructions. In this graph, each node represents an instruction and each edge represents a dependency between instructions. To make the shuffling random, instead of a breadth-first traversal, we random access the nodes that have no dependencies.

Table 4.2: Matcher table size in all architectures in LLVM on commit `860e439f`. Since GlobalIsel is a new CodeGen framework introduced in 2015, only eight architectures have implemented it.

Architecture	SelectionDAG	GlobalIsel	Architecture	SelectionDAG	GlobalIsel
AArch64	489,789	278,233	Mips	54,044	60,449
AMDGPU	493,556	338,444	NVPTX	186,134	-
ARC	1998	-	PowerPC	190,304	83,201
ARM	201,172	130,029	RISCV	2,191,899	190,009
AVR	2973	-	Sparc	6607	-
BPF	3586	-	SystemZ	53,271	-
CSKY	19,076	-	VE	71,577	-
Hexagon	178,277	-	WASM	25,991	-
Lanai	2337	-	X86	680,916	61,488
M68k	18,850	2388	XCore	3854	-
MSP430	9103	-			

4.3.2 Matcher table feedback

4.3.2.1 Matcher table instrumentation

Machine instruction generation does not correspond to compiler control flow. Different instructions can be generated by the same control flow due to the matcher table. Consequently, many patterns may not be generated yet even if edge coverage is high. To overcome this, we track the usage of the matcher table.

Similar to edge coverage, we allocate a table when the compiler starts in order to track the coverage of the matcher table. Every time an entry in the matcher table is accessed, we will record that access in our table as well.

Natively tracking the number of accesses like edge coverage is a huge memory overhead. The second and fifth column of Table 4.2 show the size of the matcher table in different architectures. The matcher tables for mainstream architectures like X86 and AArch64 have several hundred thousand entries, RISCV even has about two million entries. Natively assigning a counter to each entry like AFL++ will cost hundreds of KB of runtime memory, which will lower the fuzzing throughput [35].

Unlike control flow where the edge's execution count represents different program semantics, a matcher table entry being accessed multiple times only means the same pattern is triggered multiple

times. Therefore, we only track whether an entry is accessed or not, i.e., we use a boolean to track each entry. During instrumentation, we pack eight booleans into a byte to save space. If the table size is not a multiple of eight, we pad extra booleans.

During fuzzing, to access an entry in the matcher table, the fuzzer can calculate the offset of the entry's corresponding boolean using its index. After execution, the instrumented compiler will report a matcher table coverage back to the fuzzer. The fuzzer will use edge coverage *and* matcher table coverage together. If either table shows new coverage, we will consider the input as new.

4.3.2.2 IR mutation feedback

While the matcher table can help filter out not interesting seeds, it also contains knowledge whether an instruction or intrinsic is generated or not. We wish to pass that feedback to the mutator, so it can generate more diverse inputs. However, decoding the meaning of each entry in the matcher table is non-trivial, as LLVM hides this information when preparing the matcher table.

We first modify TableGen to dump a look-up table specifying the correspondence of matcher table entries and machine instruction patterns. The pattern reveals the condition on a specific instruction or intrinsic being generated.

Prior to fuzzing, we dump this look-up table for each architecture. During fuzzing, we will decode the matcher table coverage using the look-up table to determine which instructions haven't been generated yet. For instructions that haven't been generated, we model the condition of it; for intrinsic, we model the intrinsic as an LLVM IR function definition. Finally, we compile this information into a report and send back to the mutator to increase the chance of them been generated. This feedback is done every ten minutes, so we can provide a meaningful feedback while minimizing the runtime overhead.

4.4 Implementation

Our implementation is based on prior work FuzzMutate[127] and AFL++ [53]. FuzzMutate introduced a naive mutator with around 1000 lines of code. We add more mutation strategies for function calls, control flow graph mutation, and arbitrary data types. The mutator described in Section 4.3.1 consists of approximately 2000 new lines of C++ code, which has been contributed to the upstream LLVM's repository, augmenting the existing mutator strategies.

We implement our matcher table coverage described in Section 4.3.2.1. The implementation combines a compiler plugin to measure the size of each matcher table and insert appropriate instrumentation, along with a small runtime library to allocate and track the coverage information. It is implemented in around 1000 lines of C and C++.

We modify TableGen to dump a look-up table for each matcher table described in Section 4.3.2.2. Our decode and feedback report code is implemented in around 1000 line of C++. IRFuzzer has been open-sourced[2], the mutator has been contributed to LLVM upstream and merged to LLVM 16.

4.5 Evaluation

In order to understand how IRFuzzer helps to test LLVM code generation, we implement IRFuzzer and evaluate it. In the rest of this section, we will fuzz LLVM with different settings and tools to gain some insights to these research questions.

- **RQ1**: How does IRFuzzer compare with state-of-the-art backend fuzzers?
- **RQ2**: How does IRFuzzer compare with end-to-end fuzzers like CSmith and GrayC?
- **RQ3**: Does mutator and feedback individually contribute to IRFuzzer?
- **RQ4**: Can IRFuzzer help find new bugs in LLVM?
- **RQ5**: What are the insights we can gain from the bugs we found?

[2] `https://github.com/DataCorrupted/IRFuzzer`

The upstream LLVM repository (commit `860e439f`) currently supports 21 architectures listed in Table 4.2, excluding experimental ones. We only test on mature architectures that have a matcher table size larger than 25 000. In addition, each architecture may provide different features that can be enabled on different hardware. For simplicity, we select the backend of some popular microchips, which has a predefined set of features. These backends are widely used from user product to server applications, justifying the variety of our choice. All architectures we tested are under active development. As a result, we select 29 target CPUs[3] across 12 architectures.

We use two baseline fuzzers: AFL++ with no modification and AFL++ whose mutation module replaced with FuzzMutate. We will refer to it as FuzzMutate thereafter. All fuzzers used AFL++'s default scheduling. For fairness, we collect the seeds generated by each fuzzer and measure their branch table coverage and matcher table coverage. Branch coverage is reported by AFL++ using classical instrumentation and a default 64 KB table. Matcher table coverage is calculated as the entries accessed divided by matcher table size listed in Table 4.2.

We prepare two versions of IRFuzzer. The one labelled IRF has all designs described in Section 4.3. On the other hand, we strip all the feedback mechanism described in Section 4.3.2 and label it IRF_{bare}. When comparing with FuzzMutate, IRF_{bare}'s performance will tell us how our mutator is doing, while comparing with IRF can reveal the contribution of the feedback mechanism.

Each fuzzer process was dedicated to a single processor core on an x86_64 server with two 20-core CPUs and 692 GB of memory. Each fuzzing process lasted for one day to allow adequate exploration [100]. In addition, each experiment was repeated five times and the results were averaged to reduce random effects. To demonstrate IRFuzzer's ability to mutate IR modules and to provide a fair comparison with AFL++, each fuzzer process was initialized with 92 seed. The seeds are randomly selected from LLVM's unit testing, and they are smaller than 256 bytes to increase the throughput. The seeds are published in the artifact [137].

[3] "Target CPU" is used in LLVM to label a backend corresponding to a microchip. It can also refer to GPU or other DSP.

Table 4.3: Branch table coverage and matcher table coverage on 29 target CPUs across 12 targets in SelectionDAG. Statistics are the arithmetic mean over five trials. Bold entries are the best among baseline fuzzers. FM means AFL++ coupled with FuzzMutate, IRF means IRFuzzer

Arch	Target CPU	Branch coverage					Matcher table coverage				
		Seeds	AFL++	FM	IRF	IRF_{bare}	Seeds	AFL++	FM	IRF	IRF_{bare}
AArch64	apple-a16	59.8%	87.1%	82.9%	**94.7%**	93.9%	0.7%	1.6%	2.6%	**8.2%**	7.3%
	apple-m2	59.8%	86.9%	83.3%	**94.8%**	93.6%	0.7%	1.6%	2.6%	**8.2%**	7.2%
	cortex-a715	60.0%	87.7%	83.2%	**94.0%**	93.6%	0.7%	1.7%	2.6%	**9.4%**	7.2%
	cortex-r82	60.1%	87.0%	82.9%	**93.9%**	93.6%	0.7%	1.6%	2.6%	**7.6%**	7.2%
	cortex-x3	60.0%	93.3%	85.2%	**94.1%**	94.5%	0.7%	7.1%	2.7%	**9.1%**	7.6%
	exynos-m5	60.3%	87.4%	83.2%	**93.9%**	94.5%	0.7%	1.7%	2.6%	**7.8%**	7.5%
	tsv110	60.0%	87.3%	82.9%	**93.2%**	94.1%	0.7%	1.6%	2.6%	**7.5%**	7.3%
AMDGPU	gfx1036	70.8%	90.0%	89.1%	**96.9%**	96.7%	0.9%	2.1%	2.7%	**4.9%**	4.5%
	gfx1100	71.2%	89.7%	89.9%	**97.0%**	96.3%	1.0%	2.1%	2.9%	**4.9%**	4.5%
ARM	generic	55.5%	87.9%	82.5%	**88.6%**	88.4%	1.7%	4.3%	4.3%	**5.2%**	5.1%
Hexagon	hexagonv71t	64.8%	88.0%	86.0%	**93.6%**	93.2%	1.7%	6.6%	17.0%	**30.6%**	21.0%
	hexagonv73	64.9%	89.5%	85.7%	**94.4%**	93.4%	1.7%	7.3%	17.4%	**32.2%**	21.1%
Mips	mips64r6	52.5%	81.0%	72.7%	**85.9%**	84.5%	3.8%	10.0%	15.3%	**17.8%**	16.6%
NVPTX	sm_90	46.6%	77.5%	77.5%	**87.0%**	84.5%	1.7%	3.1%	4.7%	**26.5%**	6.2%
PowerPC	pwr9	60.3%	87.3%	86.9%	**94.5%**	94.1%	1.2%	3.6%	7.1%	**19.6%**	15.9%
RISCV	rocket-rv64	53.7%	83.0%	76.6%	**86.4%**	87.4%	1.2‰	2.0‰	2.2‰	**2.2‰**	2.3‰
	sifive-u74	54.5%	83.1%	75.9%	**86.7%**	86.6%	1.4‰	2.4‰	2.9‰	**3.0‰**	3.0‰
	sifive-x280	55.0%	84.1%	75.7%	**88.6%**	89.7%	1.4‰	2.7‰	3.1‰	**30.6‰**	31.5‰
SystemZ	z15	55.3%	84.0%	81.5%	**89.9%**	91.0%	5.2%	13.7%	27.1%	**43.8%**	38.7%
	z16	55.3%	83.7%	81.8%	**89.8%**	89.7%	5.2%	14.1%	26.5%	**43.9%**	37.6%
VE	generic	49.0%	80.4%	70.2%	**84.4%**	83.1%	3.5%	8.1%	11.4%	**14.0%**	12.6%
WASM	bleeding-edge	46.8%	**84.7%**	70.5%	82.9%	83.2%	4.1%	36.9%	10.9%	**37.6%**	35.9%
	generic	46.6%	80.2%	69.7%	**80.6%**	81.9%	4.1%	11.8%	10.6%	11.7%	11.8%
X86	alderlake	61.2%	88.0%	84.6%	**94.7%**	93.7%	0.7%	1.8%	3.1%	**8.0%**	6.2%
	emeraldrapids	60.5%	93.4%	84.4%	**93.7%**	93.5%	0.6%	12.5%	3.2%	**13.9%**	12.4%
	raptorlake	61.2%	93.5%	85.8%	**94.5%**	93.7%	0.7%	6.2%	3.3%	**8.0%**	6.2%
	sapphirerapids	60.5%	88.4%	85.4%	**93.7%**	93.2%	0.6%	1.8%	3.3%	**14.0%**	12.2%
	znver3	61.8%	86.6%	84.0%	**93.8%**	94.3%	0.7%	1.6%	3.0%	**7.7%**	6.5%
	znver4	61.0%	87.6%	84.0%	**93.7%**	94.2%	0.7%	1.8%	3.2%	**13.2%**	12.6%

4.5.1 Baseline comparison

To evaluate our strategy, we compare with two baseline implementations: AFL++ and the upstream LLVM implementation of FuzzMutate. Unmodified AFL++ lacks an LLVM IR-aware mutator, and hence we expect it to often generate invalid inputs that fail to meet the syntactic and semantic constraints of LLVM IR. On the other hand, FuzzMutate has a limited LLVM IR-aware mutator.

The result can be found in Table 4.3. The 3rd and 8th column shows the coverage brought by

initial seeds. **Bold** numbers are the best statistical significance ($p < 0.05$) when compared with other baseline fuzzers using Mann Whitney U Test.

Overall, we see AFL++ performed poorly for the purpose of testing LLVM compiler backends. In most backends, it cannot increase much matcher table coverage due to its lack of support for structured input; IRFuzzer covered more branches than AFL++ on 28 target CPUs. The output generated by AFL++ did not provide significant coverage of instruction selection patterns, as measured by the low matcher table coverage.

Both FuzzMutate and IRFuzzer reached high code coverages. FuzzMutate reached more than 75% except for generic-la64, mips64r6, generic VE, and generic WebAssembly. On the other hand, IRFuzzer performed better, achieving over 80% branch coverage for all target CPUs.

It is not sufficient to only compare branch coverage [31]. More significantly, IRFuzzer reached number one in matcher table coverage on all CPUs, indicating significantly better coverage of instruction selection patterns.

We observe that in generic WebAssembly, IRFuzzer shows no significance compared with other fuzzers. After investigation, we find that generic WebAssembly disabled many features, limiting the maximum reachable matcher table to 11.8%. This does not show that IRFuzzer is less effective on WebAssembly, as we can see that IRFuzzer still rank number one in the bleeding-edge version.

In summary, IRFuzzer achieved higher branch coverage and matcher table coverage on 28 out of 29 target CPUs. To answer **RQ1**, IRFuzzer is better at coverage when fuzzing LLVM code generation compared with state-of-the-art fuzzers.

4.5.2 Comparison with end-to-end fuzzers

In order to better understand the benefits of targeted fuzzing compared to end-to-end fuzzing, we also compare IRFuzzer with CSmith [60] and GrayC [3]. Unlike IRFuzzer, end-to-end fuzzers generates C code, which must be processed by the compiler frontend and middle-end before reaching the backend. As a result, they exercise the entire compilation pipeline, rather than focusing on just the backend. Note that although CSmith generates random, syntactically correct C code, it does not

Table 4.4: Average branch table coverage and matcher table coverage of CSmith (CS), GrayC, and IRFuzzer (IRF). `O2` and `O3` stands for different optimization levels. Bold entries are the winners.

	Arch	Branch table coverage			Matcher table coverage		
		CS	GrayC	IRF	CS	GrayC	IRF
`O2`	AArch64	94.8%	**96.1%**	93.3%	5.2%	6.9%	**7.9%**
	ARM	90.7%	**92.3%**	87.2%	4.5%	4.5%	**5.1%**
	X86	94.8%	**96.1%**	91.9%	3.5%	4.2%	**4.7%**
`O3`	AArch64	95.3%	**96.2%**	92.6%	5.4%	6.9%	**7.7%**
	ARM	91.1%	**92.5%**	86.1%	4.5%	4.5%	**4.9%**
	X86	94.9%	**96.2%**	91.5%	3.5%	4.2%	**4.6%**

implement any instrumentation and lacks feedback from executions to guide the generation process. While GrayC relies on branch coverage feedback, it does not have feedback that is customized for backends of the compilers. Besides, to test end-to-end fuzzers, we have to cross compile C to different architectures. Cross compilation itself is difficult, as we have to set up the proper tool chain for it. Therefore, we test on three most widely used architectures using generic backend.

CSmith generates C files with no initial seed. To make comparison fair, we also run IRFuzzer with **no** initial seed, since IRFuzzer is capable of generating LLVM IR from scratch. GrayC relies on deprecated APIs in LLVM 12 and can't instrument the latest LLVM, thus we download the artifact provided by GrayC [138]. The artifact consists of 715,147 C programs across ten trials. We run CSmith for 24 hours and repeat eight times, generating a total of 506,971 C programs.

We cross compile these C programs to different architectures. After compilation, we measure the resulting control-flow and matcher table coverage in the compiler backend, using the same instrumentation as IRFuzzer. `llc` default optimization to `O2`, therefore we only test `O2` and `O3`, as `O0` and `O1` are often subsets of `O2`. The results are shown in Table 4.4.

IRFuzzer achieved higher matcher table coverage on all architectures and all optimizations. Even with branch coverage feedback, GrayC is not able to generate C inputs with more matcher table coverage, further demonstrating the necessity for specialized backend fuzzing. We looked into the code generated by end-to-end fuzzers and found the reason for the coverage differences. The inferiority of matcher table coverage of end-to-end fuzzers was largely related to a lack of

coverage of vector data types. Vector instructions can only be generated when the frontend and middle-end decide a vector instruction will speed up a particular piece of code, which turns out to be rather unlikely for random C programs. However, since IRFuzzer operates on IR instructions, it can generate vector operations without relying on the frontend or middle-end of the compiler.

On the other hand, CSmith and GrayC achieved higher branch coverage. After investigation, we find that the size of generated files contributed to the difference. IRFuzzer generates bit code less than 10 KB to achieve higher throughput. Seeds generated by CSmith average to more than 40 KB after we compiled them to bit code. This means that many backend edges are executed more times, which would increase branch coverage in AFL's design. This does not show IRFuzzer's inferiority. IRFuzzer can reach fair coverage in much smaller inputs, showing the efficiency of specialized backend fuzzing.

Our evaluation answers **RQ2**, IRFuzzer achieved higher matcher table coverage than state-of-the-art end-to-end fuzzers. This shows that backend testing should not solely rely on end-to-end fuzzing, and that specialized fuzzing can improve matcher table coverage significantly.

4.5.3 Individual contributions

We are interested to know how each component helps IRFuzzer. Therefore, we strip all feedbacks in IRFuzzer to get IRFuzzer$_{bare}$. Comparison of fuzzing results can be found in Table 4.3.

Comparing IRFuzzer$_{bare}$ with FuzzMutate, we find that IRFuzzer$_{bare}$ **always** reach higher branch coverage and matcher table coverage. This means that our mutator is able to generate more diverse input than FuzzMutate. Although FuzzMutate is also a structured mutator, it lacks many advanced features we designed in Section 4.3.1. Sifive-x280 best demonstrates this improvement: on sifive-x280, IRFuzzer$_{bare}$ can cover 31.5‰ of the matcher table, while FuzzMutate only reached 3.1‰.

On the other hand, when comparing the last two columns of Table 4.3, we find that IRFuzzer is able to cover more matcher table in 26 out of 29 target CPUs compared with IRFuzzer$_{bare}$. IRFuzzer didn't show superiority on three target CPUs (rocket-rv64, sifive-x280, generic WebAssembly)

mainly because both fuzzers reached coverage ceilings allowed by that target CPU. However, we find that sometimes IRFuzzer has less branch coverage. We find the contribution of feedback mechanism is double-sided. The newly introduced matcher table coverage and feedback to mutator lowered the throughput, affecting the overall branch coverage. On the other hand, the feedback is valuable in generating more diverse inputs, contributing to higher matcher table coverage. For example, in NVPTX, IRFuzzer achieved 26.5% matcher table thanks to our feedback when IRFuzzer$_{bare}$ only covered 6.2%. Since our end goal is to test the code generation part of the backend, we believe this tradeoff is acceptable. We can answer **RQ3** confidently that both mutator and feedback design contributed to improve the matcher table coverage.

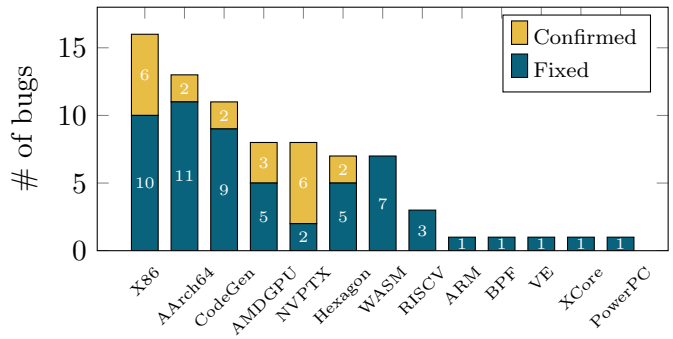
4.5.4 Bug categories and analysis

We collect all crashes found in Section 4.5.1 and Section 4.5.2. We also fuzzed other architectures with no features to extend our scope. Since GlobalIsel also uses matcher table design, we can apply IRFuzzer on it with little modification. We also fuzzed GlobalIsel for AArch64, Mips, and X86.

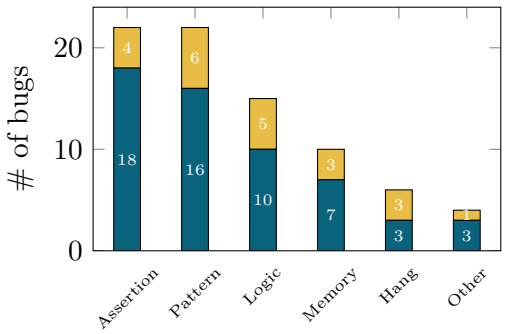
In the process, we found hundreds of crashes in the LLVM compiler. Even though these crashes all have unique stack traces, it doesn't necessarily mean they are different bugs. Some crashes have different paths but have the same root cause. Therefore, we manually analyzed all of them and report the ones we believe are bugs. In this section, we only report the bugs that have been *confirmed*. In total, IRFuzzer found 78 confirmed bugs. We manually verified that these bugs are found by IRFuzzer only and published the details anonymously [137].

These bugs are distributed in different places in LLVM codebase. Figure 4.6a shows the distribution of these bugs across LLVM. CodeGen is the library shared between architectures, meaning that a bug in CodeGen may affect all architectures. We are surprised to find that CodeGen and some widely adopted architectures have more bugs than we expected. This indicates that LLVM backend still needs more specialized fuzzing to be more fail-safe.

To better study these bugs, we categorize them into six categories: hang, memory errors, assertion failures, logic errors, missing patterns, and other bugs. Hang, memory errors and assertion



a Bugs categorized by **locations**. CodeGen refers to the code shared by all architectures, thus bugs in it can potentially affect all architectures.



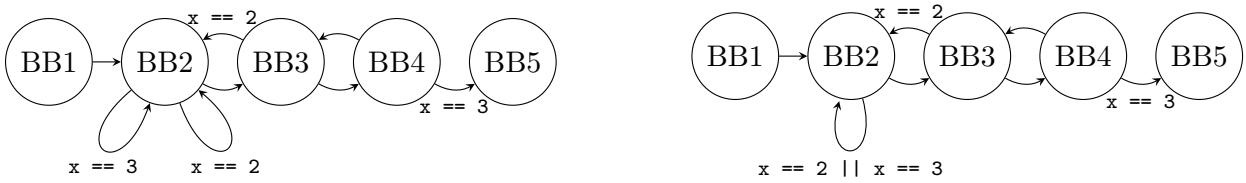
b Bugs categorized by **causes**. Most of the severe bugs are compiler hangs, memory errors, and assertion failures.

Figure 4.6: Distributions of bugs found by IRFuzzer. IRFuzzer has found **78** new bugs, **57** have been fixed.

failures are the most severe ones as they have a direct impact on end users. Missing patterns means a certain machine instruction is permitted by the hardware specification, but no matching instruction selection pattern exists. Logic error and missing patterns won't immediately affect the user, but may generate ineffective or even wrong machine instructions. Figure 4.6b shows the number of bugs in each category. Most bugs are assertion errors or missing patterns. They arise from developers' false assumption that some properties holds true during compilation, while our fuzzer proved otherwise.

We are working closely with the LLVM community to fix the bugs. 57 have been fixed, five of which are back ported to LLVM 15 as security patches. It's surprising that, despite heavy fuzzing, **all** fixed bugs are introduced before LLVM 15. This demonstrates that specialized fuzzing for compiler backend is necessary, and it provides actionable insight to developers.

We can answer **RQ4** now. We found six compiler hangs, ten memory errors, and 22 assertion failures. We also found 15 logic errors and 22 missing patterns in the matcher table. In total, we found 78 new bugs, 57 have been fixed, five have been back ported to LLVM 15 as security patches.



a Original CFG

b Optimized CFG

Figure 4.7: A piece of code we generated, simplified to CFG only. Two optimization passes are involved in this compiler hang. `TurnSwitchRangeIntoICmp` will transform Figure 4.7a into Figure 4.7b. `FoldValueComparisonIntoPredecessors` will undo the transformation, causing an infinite loop.

4.5.5 Bugs case study

4.5.5.1 Compiler hang

LLVM may execute multiple optimizations repeatedly until a fixed point is reached. However, this strategy can result in an infinite loop if not applied carefully, such as when one optimization undoes the effects of a previous optimization. Four of six hangs we found are caused by this problem.

For example, Figure 4.7 shows a simplified CFG corresponding to the code generated by IRFuzzer. This CFG will cause a compiler hang due to the interaction between two optimization passes. BB2 in Figure 4.7a consists of a `switch` statement with two self loop edges. The `TurnSwitchRangeIntoICmp` optimization attempts to rewrite the condition as a branch predicate because `x == 2 || x == 3` can be optimized using bit operations, rewriting Figure 4.7a into Figure 4.7b. However, the `FoldValueComparisonIntoPredecessors` optimization converts this code back into a `switch` statement to reduce the number of comparison operations, turning the CFG back to Figure 4.7a. As a result, a fixed point is never reached, creating an infinite loop.

This bug is hard to trigger since the bug can only be triggered when the `switch` in Figure 4.7b has exactly two destinations (BB2 and BB3), and the switch conditions are consecutive, enabling the `TurnSwitchRangeIntoICmp` optimization. This combination is unlikely to be created during manual testing, and can only happen through the interaction of two largely unrelated pieces of code. Yet, we are able to discover this catastrophic combination through our CFG mutation strategy in a time frame amenable to run fuzzing on every nightly build with little human intervention.

In this case, we answer **RQ5** by advising the developers to carefully read code that may modify the same location before they push out a new optimization that changes the code.

4.5.5.2 Memory error

LLVM often hides memory management from the developers so they don't have to manually manage it. Still, we found five memory errors: one null pointer dereference, two double frees and two out of bounds (OOB) accesses. After inspection, we determined that both double free bugs were indirectly caused by OOB accesses, which didn't result in crash immediately. All four OOB accesses are caused by developers using constants from the program under compilation without validation. In these cases, an OOB access or undefined behavior in the program being compiled were able to crash the compiler itself.

As an example, we consider a case involving vector types, shown in Listing 4.5. LLVM natively supports to enable SIMD optimizations. The `insertelement` IR instruction instruction inserts a value into a vector at a specified index. If the given index is out of the bounds, the behavior is undefined. The compiler can leverage this to remove the `insertelement` instruction.

In most cases, LLVM's module verifier checks for invalid indices, and will reject any negative indices before compilation. As a result, developers may implicitly assume that indices in this code are *always* non-negative. However, the module verifier does allow `undef` as a valid index. `undef` in LLVM represents a value that can be anything, and is represented as an index `IntImm` of -1 at this point in the code, resulting in an OOB access at Line 7.

This example shows that our IR instruction mutation method is better than high-level language generation, since `undef` is not a primitive high-level language construct and will only be introduced through other optimizations. By generating LLVM IR directly, we have more control over instruction operands and can directly generate values like `undef`. To answer **RQ5**, we recommend developers to be careful about uncommon values like `undef` and `poison`, which may appear during compilation.

```
1  bool CombinerHelper::matchCombineInsertVecElts(...){
2    while (...) {
3  +   if (IntImm >= NumElts || IntImm < 0)
4  -   if (IntImm >= NumElts)
5        return false;
6      if (!MatchInfo[IntImm])
7          MatchInfo[IntImm] = TmpReg;
8      CurrInst = TmpInst;
9      ...
10   }
11 }
```

Listing 4.5 A snippet of code in AArch64 where the index (`IntImm`) is not sanitized before usage. This diff is our patch to fix this bug.

```
1  bool IRTranslator::translateExtractElement(
2    const User &U,
3    MachineIRBuilder &MIRBuilder)
4  {
5      Register Idx;
6      const LLT VecIdxTy = LLT::scalar(PreferredVecIdxWidth);
7      Idx = MIRBuilder
8  -           .buildSExtOrTrunc(VecIdxTy, Idx)
9  +           .buildZExtOrTrunc(VecIdxTy, Idx)
10             .getReg(0);
11     ...
12 }
```

Listing 4.6 A snippet of code in LLVM where index of a vector is treated as signed value. This diff is our patch to fix this bug.

Figure 4.8: Two bugs we found in LLVM codebase. Both of them will lead to compiler crash and have been fixed.

4.5.5.3 Logic error

Logic error usually starts with unclear documentation or undocumented assumptions. Middle-end and backend are developed by different programmers, who may interpret ambiguous documentations differently.

For example, Listing 4.6 shows a bug we found in the LLVM backend. When translating the IR instruction `extractelement`, the index is extended as a signed integer. The code translates constants like `char 255` into `-1`. This bug generates incorrect machine instructions and affects the LLVM backend for seven architectures.

The bug was introduced in LLVM nine years ago and was never noticed for several reasons. First, it is less common for compiler frontends to generate vector operations, as we have seen in

Section 4.5.2, and even more rare to use an index that is large enough to wrap around to a negative integer. However, more importantly, we discovered that the documentation was ambiguous with respect to the desired behavior. The documentations indicated that "The index may be a variable of any integer type" without giving more details on how it should be interpreted. Therefore, when this bug was introduced nine years ago, it was actually compliant with the incomplete documentation at the time. This exemplifies how complex software interfaces can be incompletely specified, which further justifies our specialized fuzzing. In this case, we have fixed the bug and updated the documentation to reflect the intended interpretation of the index as an unsigned integer. To answer **RQ5**, we encourage developers of LLVM to communicate more on the documentations. Review and update documentations with the development of the code would be a good practice.

4.6 Related work

Prior work has focused on compiler testing [139–141]. One popular approach is to generate inputs for compilers to compile. Purdom[142] generates program based on context free grammar. Superion[135] and Nautilus[55] also relies on context free grammar for fuzzing. However, context free grammar based methods cannot generate semantically meaningful programs. These efforts are effective in testing frontend parsers, but cannot reach the backends effectively.

While many fuzzers are testing the frontend of the compiler using grammar based method [143], some work also tests the correctness of middle-end [2, 119, 144, 145]. To the best of our knowledge, IRFuzzer is the first one to verify the compiler backend using an architecture independent method.

Some work does end-to-end tests using high-level programming languages. CSmith[60], YARPGen[121], and Grayc [3] generate C and C++ programs. AI has also been used for program generate for the purpose of compiler testing [113–115]. However, end-to-end testing implies that there is a need to create a generator for every language, like JavaScript [56, 136], Rust [146], and Java [147–149]. POLYGLOT[150] introduced a language-free IR and mutator based on it. Most fuzzers have no feedback from the compiler. Even though Grayc [3] introduced branch coverage feedback, it was

unable to trigger backend bugs due to language limitations and compiler optimizations discussed in Section 4.5.5.3. Instead of directly generating a program, Equivalence Modulo Inputs [122, 151, 152] mutates an existing C program to preserve its semantics. Therefore, the program before and after mutation should have the same behavior. Combining CSmith and EMI, Lidbury et al. mutate program to test OpenCL compiler [153]. However, the language limits these work, since the generator cannot help when the language frontend cannot exercise a feature in the compiler.

Formal verification is another valuable part of compiler verification [154]. Verasco [124] is a formally verified C analyzer. CompCert [155] is a compiler for a subset of C that is formally verified. There is work that verifies other languages, like Rust [156] and Lustre [125]. However, formal verification cannot scale to large compilers like LLVM, therefore it has a limited impact in the community.

There is also work on intermediate representation generation. FuzzMutate directly generates LLVM IR[127]. However, FuzzMutate has no feedback unless combined with fuzzers like AFL++[53] or libFuzzer[86]. Some work focus on testing of a specific compiler [157, 158]. Tzer focuses on IR mutation in the context of a tensor compiler [158]. However, Tzer relies on LLVM's Coverage Sanitizer that only tracks code coverage. Similar to IRFuzzer's approach, ClassMing directly mutates on Java byte code[159]. Neither Tzer nor ClassMing designed a feedback method that can apply to LLVM's scenarios.

Chapter 5

Understanding programs by exploiting fuzzer generated test cases

5.1 Introduction

Code intelligence powered by machine learning has attracted considerable attention in both the AI and software engineering community. Particularly, code representation learning, which aims to encode functional semantics of source code, lays the foundation for achieving the intelligence and is of great interest. The learned representation can be applied to various downstream tasks, including code classification [67], code summarization [65], clone detection [66, 67], etc.

Many efforts inspired by the developments of natural language understanding have been devoted to learning code representations, among which it has been increasingly popular to adopt large language models (LLMs) that are capable of learning contextual information from data at scale [160, 161]. The LLMs can then be fine-tuned on domain-specific code to achieve superior performance compared with tradition models.

Despite being effective, these natural language processing methods do not fit perfectly for handling programs. Specifically, programs are heavily structured and syntax-strict (to be understood by compilers or interpreters), while natural language corpus is not. As basic units of programs,

functions and subroutines can take a variety of argument values to demonstrate different logical behaviors or return different results. That being said, the relationship between inputs and possible outputs/behaviors essentially represents the functions/subroutines and further the whole programs.

In this chapter, we propose to incorporate such a relationship into learning for a deeper understanding of programs. In fact, given enough inputs to execute all pieces of the code, then the outputs would include enough runtime information we need to profile and understand the program. However, it is nontrivial to generate a limited number of inputs that are representative enough to execute every part of the code. Without a proper strategy, we may end up with a large number of inputs that execute similar parts of codes. To address the issue, we opt to utilize fuzz testing (also known as fuzzing) [162], which is a common software testing practice and dynamic analysis tool whose original goal is to find software bugs by executing as much code as possible. More specifically, we repurpose fuzz testing to generate input and output data for assisting code representation learning, and we demonstrate how the input and output data (i.e., test cases) can be appropriately incorporated into existing LLMs to achieve superior program understanding performance.

The contributions of this chapter are three-fold. First, by recognizing the essence code representation learning, we propose to take advantage of the relationship between inputs and possible outputs for achieving a deeper understanding of programs. Second, we, for the first time, repurpose fuzz testing to assist code presentation learning, marrying these two concepts from different communities for achieving more powerful AI. Third, we obtain state-of-the-art results on typical program understanding tasks including clone detection and code classification, in comparison to prior arts.

5.2 Related work

In this section, we introduce related work on code understanding (from the natural language understanding community) and fuzzing (from the software engineering community).

5.2.1 Code representation learning

Inspired by the success of LLMs in natural language processing [163–165], LLMs trained on programming languages have also been widely used to drive code intelligence. For instance, [166] proposed cuBERT to train BERT models on a curated and deduplicated corpus of 7.4M Python files from GitHub, and adapt the pre-trained models to various code classification tasks and a program repair task. Thereafter, a bunch of methods have been developed and a variety of LLMs have been trained on code data, including CodeBERT [160], CodeT5 [167], and CodeGPT [168].

The importance of comprehending syntax and structures for learning code representations has also been pointed out by several prior arts, and methods that incorporate programming-language-specified features, including abstract syntax tree [169, 170], control or data flow graphs [171], and intermediate representation of code [172] have been developed. These methods only utilize information available for static analysis. It is generally difficult for static analysis to be both safe and sound [173] when analyzing the behavior of programs. For instance, a path that exists on the control flow graph may never be executed due to data-flow limitations. Our work is the first to take dynamic program information (by generating and exploiting test cases with inputs and outputs) into account for code representation learning. An input will lead to execution of part of the program and an output (or some behaviors if no output is required), which would reflect the functionality of that part of the program. We hypothesize that if we have enough inputs to execute the code sufficiently, the outputs would also include enough runtime information that we need to profile the program.

5.2.2 Fuzzing

Fuzz testing, or fuzzing, is a process that tests the correctness of programs. Fuzzing can be roughly considered as a four-stage loop. First the program is executed with a given input. Second, the behavior of the program is monitored to determine if any new behavior is triggered. Third, if a new behavior is present, the corresponding input will be saved into a store, otherwise, the input is discarded as not interesting. Finally, a mutator takes a saved input in the store, mutates it in different fashions and sends the input for another round of execution.

American Fuzzy Lop (AFL) [1] is the first fuzzer to implement behavior monitoring using branch coverage. It tracks which edges of the control flow graph have been executed. Since the invention of AFL, many innovations have been made to improve the overall fuzzing performance. [31, 40] modified branch coverage to lower the overhead while improving tracking sensitivity. [38] proposed that power scheduling is better than a first-in-first-out queue for input store and improved the fuzzing performance by a magnitude. [44] introduced new mutation algorithms and showed superior performance than random mutation. Many of the changes have been incorporated into a more modern tool called AFL++ [53].

Unfortunately, current use of fuzzers only focuses on the bugs in the software [19, 55, 107] and did not show possibility of adopting fuzzing results in code representation learning for AI. We identify that these results can be used to profile programs and improve the performance of code representation learning and program understanding.

5.3 Method

As mentioned in Section 5.1, programs show strict syntax. To inspire deeper understanding of the syntax and logical behaviors of a program or functions/subroutines (which are the building blocks of the program), we attempt to exploit the relationship between their inputs and possible outputs/behaviors for achieving improved understanding of programs and code, akin to how engineers understands third-party code.

However, with existing learning techniques, it seems nontrivial to generate inputs that could lead to execution of sufficient part of the code, thus we resort to fuzzing to achieve this goal.

5.3.1 Fuzzing for obtaining inputs and outputs

Despite being widely adopted for testing software, fuzzing has rarely been adopted in machine learning tasks. In general, fuzzing is a software testing practice, whose goal is to find software bugs by executing as much code as possible. To achieve this, it executes the program with different inputs

and monitors the behavior of each execution. Therefore, as byproducts of fuzzing, a large number of inputs may be produced by a fuzzer, each triggering a new behavior of the program under test.

Fuzzing is programming language agnostic in general. However, with only source code, we have to compile the programs into executable files for fuzzing. We mainly describe details for four mainstream languages (C, C++, Java, and Python), and a tool was specifically designed to build the programs for fuzzing. This tool interacts with the compiler or interpreter to automatically fix some problems that prevent it from being fuzzed. Since the main aim of this work is to assist models to better understand programs, we fix problems that do not affect the semantics and functionality of code but prevent fuzzing.

For C and C++, we treat them as C++ files. Some semantics-irrelevant errors in the program would prevent the code from compiling and fuzzing. For example, missing headers, absent return of a `main` function that is defined to have one, and misuse of reserved keyword. In order to fix these compilation errors, we designed a compiler plugin that can automatically fix these. First we run the lexer and parser on the program to gain abstract syntax tree, which would make code transformation much easier. Then we designed a parser to parse error message from the compiler. We introduce several fixes to correct the program for different errors.

1. **Missing headers.** We included most commonly used headers in the C++ library at the head of each program.

2. **Incorrect return type and/or arguments.** For instance, if a `main` function is defined as "`int main()`" but provides no return, we fixed it by added "`return 0;`" to the end of the program.

3. **Misuse of keywords in the standard library.** Reserved keywords might be misused as variables and we added a "`fixed_`" prefix to each of such variables to avoid keyword violation.

4. **Incorrect `struct` definition.** Many structures were defined without a semicolon after it, we will append that semicolon.

5. **Undeclared identifier.** We notice that many programs use static values as a constant value,

yet the value is sometimes missing. We would analyze the usage of the constants and insert definitions for them.

For Java programs, we compiled them into bytecodes using Kelinci [174] to instrument them for fuzzing. Not all programs we tested were named `Main.java` but they all defined a `Main` class. In order to compile them, we changed the `Main` class to its file name in order to compile it. For each program, a TCP server was added to communicate with a fork server which then sends data to the fuzzer.

Python is the most difficult language. First many lexical errors are not as easy to fix as C/C++ and Java. For example, if the program mixed tabs and spaces, it is hard to infer what is the intended indention. To solve this, we used `autopep8`[1] to transform the program. The next challenge is that Python2 and Python3 can't be easily distinguished, therefore, it is unclear which interpreter should be used for fuzzing. To detect the version, also to verify the correctness of the transformation, we treated all code as Python3 in the first place and try to compile python program to bytecode using `py_compile`. If the compilation failed, then it was probably a Python2 implementation and we tried to convert it to Python3 using `2to3`[2]. Finally, we had to instrument the behavior monitoring and reporting to communicate with the fuzzer. We used `py-afl-fuzz`[3] to achieve this.

We want to point out that all the changes made in this section are for fuzzing only. When training models in the following sections, the programs remain unchanged.

We selected AFL++ [53] as our fuzzer and fuzzed all experimental data on a server with 2 20-core 40-thread x86_64 CPUs and 692GB of memory. Each fuzzer only has one thread and ran until it exhausts all paths or a K-minute timeout is triggered. The stored inputs that are of interest to AFL++ can then be utilized to execute the program and obtain outputs, and they constitute the fuzzing test cases. The test cases (i.e., pairs of inputs and outputs) were produced in bytes and we may decode it into human readable strings.

[1] `https://pypi.org/project/autopep8/`
[2] `https://docs.python.org/3/library/2to3.html`
[3] `https://pypi.org/project/python-afl/`

5.3.2 Model

Although it is possible to train representation learning models from scratch using the obtained fuzzing test cases, it can be more effective to take advantage of previous pre-training effort. In particular, given a pre-trained LLM, we attempt to take these test cases as model inputs somehow. Considering that the LLM was mostly trained on programming language and natural language corpus [160, 167, 170], the source code of the program is fed into the model together with fuzzing test cases, by concatenating the two parts.

5.3.3 Prompting

The fuzzing test cases in their raw format are a series of bytes, and, by decoding, we can obtain a series of Unicode strings which are unorganized. To help LLMs better understand these test cases, we introduce cloze prompts [175]. Prompt have shown significant power in natural language processing since the invention of LLMs. Considering that LLMs can be pre-trained on both natural language corpora and programming language corpora, we design both natural-language-based prompts and programming-language-based prompts for each pair of input (denoted by `[INPUT]`) and output (denoted by `[OUTPUT]`) as follows:

1. **Natural-language-based prompt:**

   (a) `[SEP]` + "input: " + `[INPUT]` + "," + "output: " + `[OUTPUT]`;

   (b) `[SEP]` + "input is " + `[INPUT]` + "and" + "output is " + `[OUTPUT]`;

2. **Programming-language-based prompt:**

   (a) `[SEP]` + "`cin>>`" + `[INPUT]` + "`;`" + "`cout<<`" + `[OUTPUT]`; (For C/C++)

   (b) `[SEP]` + "`System.in` " + `[INPUT]` + ";" + "`System.out`" + `[OUTPUT]`; (For Java)

   (c) `[SEP]` + "`input()`" + `[INPUT]` + "\n" + "`print`" + `[OUTPUT]`; (For Python)

In experiments, we found that the programming-language-based prompts are more effective and we will stick with it in the sequel of this chapter, if not specified. This is unsurprising since the

Table 5.1: Dataset statistics.

Dataset	# of problems	# of programs
POJ104	104	52K
C++1000	1000	500K
Python800	800	240K
Java250	250	75K

Method	Java250	Python800	C++1000*
Rule-based w/SPT(AROMA) [68]	19.00	19.00	-
GNN w/SPT(MISIM) [68]	64.00	65.00	-
CodeBERT+FineT [160]	81.47	83.23	44.94
UniXcoder+FineT [170]	84.35	85.00	49.75
CodeBERT+FuzzT (ours)	**83.39**	**85.64**	**54.92**
UniXcoder+FuzzT (ours)	**86.74**	**86.01**	**60.21**

Table 5.2: Clone detection results on CodeNet. Compared with normal fine-tuning (FineT), our fuzz tuning (FuzzT) leads to significant improvements and new state-of-the-arts. C++1000* contains 16% of all problems, which is a roughly 6.3x downsample of the original dataset (see Table 5.8 for results on other scales). Bold stats are better.

fuzzing test cases can stay in harmony with the source code with such a prompt.

Each prompted pair of input and output can be concatenated together before being further concatenated with the source code. Pre-trained LLMs can be tuned on downstream datasets with their inputs being modified to consider both the source code and fuzzing test cases. We call this method *fuzz tuning* in the chapter.

5.4 Experimental results

In this section, we report experimental results to verify the effectiveness of our fuzz tuning. We consider popular tasks (i.e., clone detection and code classification) and datasets involving mainstream languages including C, C++, Java, and Python. Experiments were performed on NVIDIA V100 GPUs using PyTorch 1.7.0 [176] implementations.

Datasets. Our experiments were performed mainly on two datasets, i.e., POJ104 [67] and

Table 5.3: Clone detection results on POJ104. Our fuzz tuning (FuzzT) leads to state-of-the-art results. Bold stats are better.

Method	MAP@R
CodeBERT+FineT [160]	84.29
GraphCodeBERT+FineT [171]	85.16
PLBART+FineT [177]	86.27
SYNCOBERT+FineT [178]	88.24
CodeT5+FineT [167]	88.65
ContraBERT+FineT [179]	90.46
UniXcoder+FineT [170]	90.52
CodeBERT+FuzzT (ours)	**92.01**
UniXcoder+FuzzT (ours)	**93.40**

CodeNet [68]. POJ104 has been incorporated into CodeXGlue [168] and is widely used. It consists of 104 problems, each containing 500 C/C++ implementations. CodeNet is a recently proposed large-scale dataset for AI for code applications, and it contains programs written in C++, Java, and Python. In particular, it has four subsets for these languages: Java250, Python800, C++1000, and C++1400. We chose Java250, Python800, and C++1000 for experiments, which cover all the three languages in CodeNet. Java250 consists of 250 problems where each includes 300 Java programs, and Python800 consists of 800 problems where each includes 300 Python programs. C++1000 consists of 1000 problems where each includes 500 C++ programs, and it is mainly used to verify the effectiveness of our method over various training scales (i.e., our fuzz tuning will be performed on various subsampling ratios of the set) in this chapter. See Table 5.1 for a summarization of key information of all datasets.

Pre-trained LLMs. To make our experiments more comprehensive, our fuzz tuning (FuzzT) was tested on two different LLMs: CodeBERT [160] and UniXcoder [170] that were pre-trained on both natural languages and programming languages.

Obtaining test cases. We set $K = 5$ for the fuzzer. In POJ104, 90.3% of all fuzzed programs quits before timeout, which justifies our decision. Taking advantage of our effort in Section 5.3.1, all datasets have more than 95% of the programs compiled / validated, and all of them have more

Method	Java250	Python800	C++1000†
GIN [68]	6.74%	5.83%	-
CodeGraph+GCN [68]	5.90%	12.2%	-
C-BERT+FineT [68]	2.60%	2.91%	-
CodeBERT+FineT [160]	2.37%	2.19%	16.34%
UniXcoder+FineT [170]	2.02%	1.95%	14.57%
CodeBERT+FuzzT (ours)	**1.77%**	**1.61%**	**7.63%**
UniXcoder+FuzzT (ours)	**1.56%**	**1.28%**	**6.18%**

Table 5.4: Code classification results on CodeNet. Our fuzz tuning (FuzzT) leads to new state-of-the-arts. C++1000† contains 40% of all problems, which is a 2.5x downsample of the original dataset (see Table 5.9 for results on other data scales). Bold stats are better.

Table 5.5: Code classification results on POJ104. Our fuzz tuning (FuzzT) leads to new state-of-the-arts. Bold stats are better.

Method	Error Rate
TBCNN [67]	6.00%
ProGraML [180]	3.38%
OSCAR [172]	1.92%
CodeBERT+FineT [160]	1.61%
UniXcoder+FineT [170]	1.61%
CodeBERT+FuzzT (ours)	**1.40%**
UniXcoder+FuzzT (ours)	**1.38%**

than 90% of programs fuzzed.

5.4.1 Clone detection results

The clone detection task aims to measure the similarity between two code snippets or two programs, which can help reduce bugs and prevent the loss of intellectual property.

Given a code snippet or the source code of a program as a query, models should detect semantically similar implementations on the test set. POJ104 is adopted as the default dataset for code clone detection on CodeXGlue, thus we did experiment on it first. We followed previous work [168] and used a 64/16/24 split. That said, training was performed on 64 problems while validation and test were performed on other 16 and 24 problems, respectively. Besides, We further experimented on

CodeNet which shows a larger data scale and variety. Java250, Python800, and C++1000 were used, and we followed [68] which used a 50%/25%/25% split for training/validation/test for all these concerned sets. C++1000 is mainly used to test our method over various training scales in Section 5.4.4 with the full test set, and we will only discuss results on the smallest subsampling ratio (which is roughly 6.3x, achieving by randomly selecting 16% of the problems for experiments) in this subsection. We denoted such a subsampled set as C++1000*. All models tested here were tuned on a single V100, for no longer than 8 GPU hours.

Results were evaluated using the mean average precision@R (MAP@R) [181]. To train our models, we followed previous work [168] and directly set the learning rate, batch size, and maximal sequence length (for code tokens) to 2e-5, 8, and 400, respectively. We used the Adam optimizer [182] to fine-tune each pre-trained model for 2 epochs. The best model on the validation set is selected to test. We adopted the same hyper-parameters on both POJ104 and CodeNet.

Table 5.3 provides the results on POJ104. It is obvious that, when the proposed fuzz tuning is applied, we obtain significant performance gains with both CodeBERT and UniXcoder. More specifically, comparing with the normal fine-turning (FineT) method, we obtained **+7.72%** and **+2.88%** absolute gains in MAP@R with CodeBERT and UniXcoder, respectively. Such practical improvements clearly demonstrate the benefits of incorporating fuzzing test cases into program understanding. In addition, fuzz tuning obtained models (i.e., CodeBERT+FuzzT and UniXcoder+FuzzT) outperform all other state-of-the-art models significantly, leading to obvious empirical superiority (i.e., **+2.94%**) comparing even with a very recent winning solution on the CodeXGLUE benchmark [4] called ContraBERT.

Table 5.2 demonstrates the results on CodeNet. Apparently, on CodeNet, our fuzz tuning also leads to significant performance gains on CodeBERT and UniXcoder, when compared with FineT.

[4] `https://microsoft.github.io/CodeXGLUE/`

5.4.2 Code classification results

The concerned code classification task [67] requires that we assign the same label to programs that were implemented to solve the same problem and achieve the same goal. The experiments were also performed on POJ104 and CodeNet, where each unique problem is considered as a single class. For POJ104, we adopted the same dataset split as in [172]'s work, and, for CodeNet, we kept the official data split [68]. As previously mentioned, C++1000 in CodeNet is mainly used to test our method over various training scales in Section 5.4.4, and we will only discuss results on the smallest subsampling ratio (which is 2.5x, achieving by randomly selecting 40% of the programs for experiments) here. We denoted such a subsampled tuning set as C++1000†. We followed the hyper-parameter setting of CodeBERT in defect detection to set a learning rate of 2e-5, a training batch size of 32, and a maximal sequence length (for code tokens) of 400. We tuned pre-trained LLMs for 10 epochs and selected the models that performed the best on the validation set and report their results on the test sets. Error rate of different methods are reported for comparison. All our code classification models were were tuned on two V100, for no longer than 8 hours.

Table 5.5 and Table 5.4 summarize the results. Similarly, We observe that our fuzz tuning bring significant improvement, comparing with the normal fine-tuning (FineT) method, it leads to **+0.21%** and **+0.23%** absolute performance gain in reducing the error rate with CodeBERT and UniXcoder, respectively, on POJ104. Fuzz tuning obtained models (i.e, CodeBERT+FuzzT and UniXcoder+FuzzT) also outperform all previous models on this task on POJ104, leading to new state-of-the-arts. The same conclusion can also be drawn on CodeNet, showing that the effectiveness of our method hold on various programming languages.

The results on both clone detection and code classification demonstrate the effectiveness of our fuzz tuning. Both tasks requires the model to understand not only the structure of the code, but further the semantics, which is hard to acquire by simply looking at the code. Yet, provided with inputs and outputs, the model can excel. We contribute this accuracy gain to program profiling provided through fuzzing. These profiles include essential dynamic information that isn't used by any other models.

Table 5.6: Comparing using raw and decoded fuzzing test cases in tuning clone detection (CD) and code classification (CC) models on POJ104. MAP@R and the error rate are evaluated for the two tasks, respectively. Bold stats are better.

Decoding	CD	CC
CodeBERT+FineT	84.29	1.61%
CodeBERT+FuzzT		
-in bytes	84.21	1.55%
-in UTF-8 string	**92.01**	**1.40%**

Table 5.7: Comparing different prompts for our fuzz tuning on the clone detection (CD) and code classification (CC) tasks on POJ104. Bold stats are better.

Prompt Type	CD	CC
CodeBERT+FineT	84.29	1.61%
CodeBERT+FuzzT		
-w/o prompt	89.36	1.51%
-NL prompt, type (a)	91.14	1.54%
-NL prompt, type (b)	91.59	1.58%
-PL prompt, for C/C++	**92.01**	**1.40%**

5.4.3 Ablation study

In this subsection, we investigate impacting factors in our method: including the quality of test cases, decoding, and prompting.

Random cases vs fuzzing cases. Given the success of fuzz tuning in clone detection and code classification, the effectiveness of incorporating test cases can be recognized. One may expect that random input generator can work to some extent, for providing test cases. Unfortunately, our evaluation shows otherwise. We tried following this idea and crafted around 2000 inputs for each program, yet none of them is valid and understandable to the program. This result is expected, since the chance of a byte being a digit is only 10/256, there is less than 1% change of generating a 3-digit number. Thus, it is reasonable to conclude that random input generator is prone to generating invalid inputs, which lead to crash and hang of the program and cannot be used to profile it. By contrast, our fuzzer provides behavior monitoring, all these ineffective inputs are filtered and not reported in the first place.

Table 5.8: How different methods scale with the size of training/tuning dataset on the C++1000 *clone detection* task. Bold stats are better.

Method	4% (↓25x)	8% (↓12.5x)	16% (↓∼6.3x)
CodeBERT+FineT	26.92	36.79	44.94
CodeBERT+FuzzT	**30.66**	**40.57**	**54.92**
UniXcoder+FineT	34.71	43.06	49.75
UniXcoder+FuzzT	**42.53**	**51.83**	**60.21**

Decoding. As mentioned in Section 5.3.1, the fuzzer processes obtained inputs as a series of bytes. We argue that reading test cases as bytes will cause severe performance degradation, since LLMs are pre-trained using human-readable codes and natural languages, which explains why we decode the obtained bytes before feeding them to LLMs. To verify the effectiveness of decoding, we compare using human-readable UTF-8 strings and those raw bytes, both with cloze prompts, for program understanding. The experiment was conducted on the POJ104 clone detection task and the POJ104 code classification task. Table 5.6 shows the results. Apparently, human-readable test cases perform much better than bytes-format ones on both two tasks.

Prompting. We then compare the performance of fuzz tuning with and without prompting. Table 5.7 demonstrates the results. For prompting, two types of natural-language-based (NL-based) prompts and the advocated programming-language-based (PL-based) prompt are tested. Apparently, prompting is beneficial. As has been mentioned in Section 5.3, the PL-based prompt outperforms the two types of NL-based prompts. It shows a **+2.65%** absolute gain on the POJ104 clone detection task and a **+0.11%** absolute gain on code classification, compared with an implementation of fuzz tuning without prompts. For clone detection, prompting is always effective, no matter it is NL-based or PL-based, while, for code classification, the NL-based prompts fail.

5.4.4 Data scale

We then investigate whether our fuzz tuning is effective on various training data scales. To achieve this, we subsampled from C++1000 in CodeNet to construct data sets of various scales to perform fuzz tuning. The official split of C++1000 was considered to construct test sets [68], and

Table 5.9: How different methods scale with the size of training/tuning dataset on the C++1000 *code classification* task. Bold stats are better.

Method	10% (↓10x)	20% (↓5x)	40% (↓2.5x)
CodeBERT+FineT	34.38%	19.15%	16.34%
CodeBERT+FuzzT	**30.90%**	**12.53%**	**7.63%**
UniXcoder+FineT	21.66%	16.24%	14.57%
UniXcoder+FuzzT	**14.48%**	**8.39%**	**6.18%**

the same test sets were adopted for testing models obtained on all these training scales. For clone detection, we sampled 4%, 8%, and 16% of the training and validation problems (i.e., subsampled the training and validation set by 25x, 12.5x, and roughly 6.3x). For the code classification task, we sampled 10%, 20% and 40% (i.e., subsampled by 10x, 5x, and 2.5x) of the training and validation and keep the sample ratio between the two sets as 4:1. We adopted the same experimental settings as in Section 5.4.1 and Section 5.4.2.

Results of different fuzz tuning scales are provided in Table 5.8 and Table 5.9. Apparently, our fuzz tuning is effective on all these training scales. In particular, for clone detection, when only 4% of the data is used for training, normal fine-tuning of CodeBERT and UniXcoder shows an MAP@R of only 26.92 and 34.71, respectively, while, by introducing fuzz tuning, we can achieve 30.66 and 42.53, respectively, showing even more obvious superiority than with 16% of the data. It is also possible for both fine-tuning and fuzz tuning to scale to more than 16% of the data, yet it requires more than 10 epochs to reach their performance plateaus and weaken the necessity of pre-training, thus we will leave it to future work for exploration. The same conclusion can be drawn for code classification.

5.4.5 Case study

In this section, we extract some real cases in the concerned dataset (i.e., POJ104) to show how our fuzz tuning works. Figure 5.1 reports the achieved per-program MAP@R and the performance gap between FuzzT and FineT on the POJ104 test set, *with CodeBERT*. We see that FuzzT outperforms FineT on 17 out of the 24 test problems.

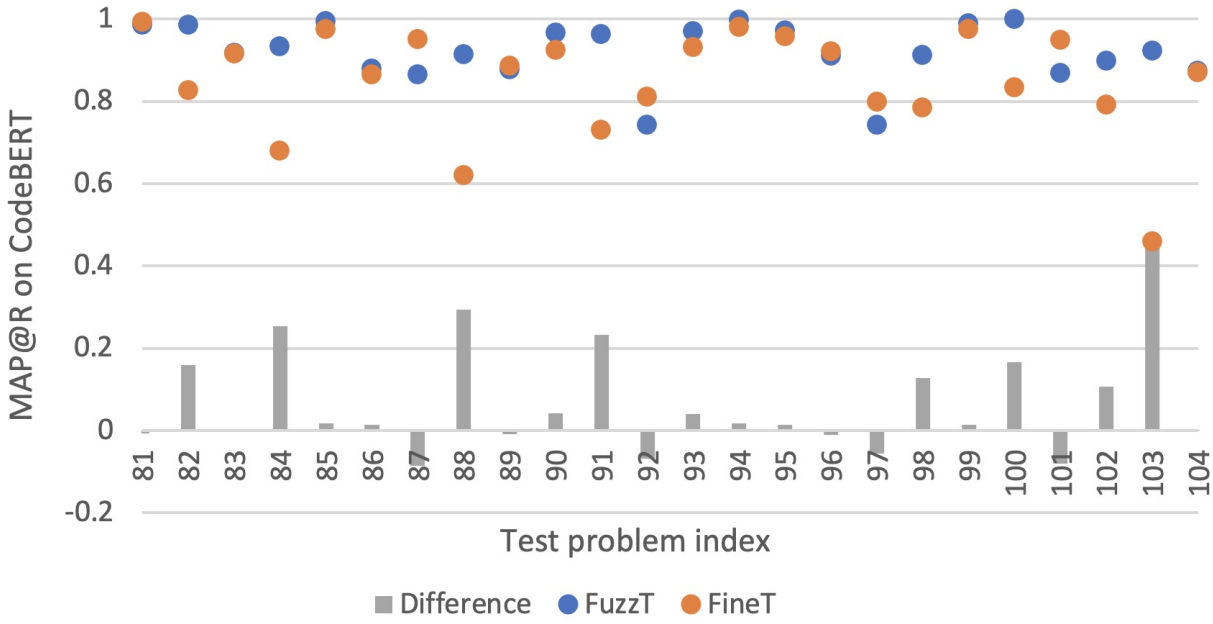


Figure 5.1: Per-problem clone detection performance on the POJ104 test set, using *CoderBERT+FineT* or *CoderBERT+FuzzT*. The horizontal axis shows the ID of the POJ104 problems, and the vertical axis is the MAP@R.

Figure 5.1 demonstrates that using the normal fine-tuning leads to very low MAP@R on Problem 103 of POJ104 [5], yet our fuzz tuning more than doubled the score. Although POJ104 does not describe each problem in detail, we did some investigations and conjecture that this particular problem is asking how many identical consecutive letters are there in a given string, if letter case is ignored. Our investigations show that many programmers all unifies the letter case in the string first, but they disagree on whether to use uppercase letters or lowercase letter sand disagree on how to achieve this, leading to different implementations including utilizing standard library calls (i.e., to convert each character "c" using "`toupper(c)`"), calculating offset by casting (i.e., implementing something like `c-'A'+'a'`), and static mapping (i.e., using "`caseMap[c]`"). This will pose challenges to models for understanding their functionality, if fine-tuning on source code only. One may expect this particular problem to be addressed by pre-training on relevant data or by taking more advantage of static information of programs. This is possible, since for other pre-trained LLMs, Problem 103

[5]Note that Problem 1-80 are training and validation problems, and Problem 81-104 are test problems.

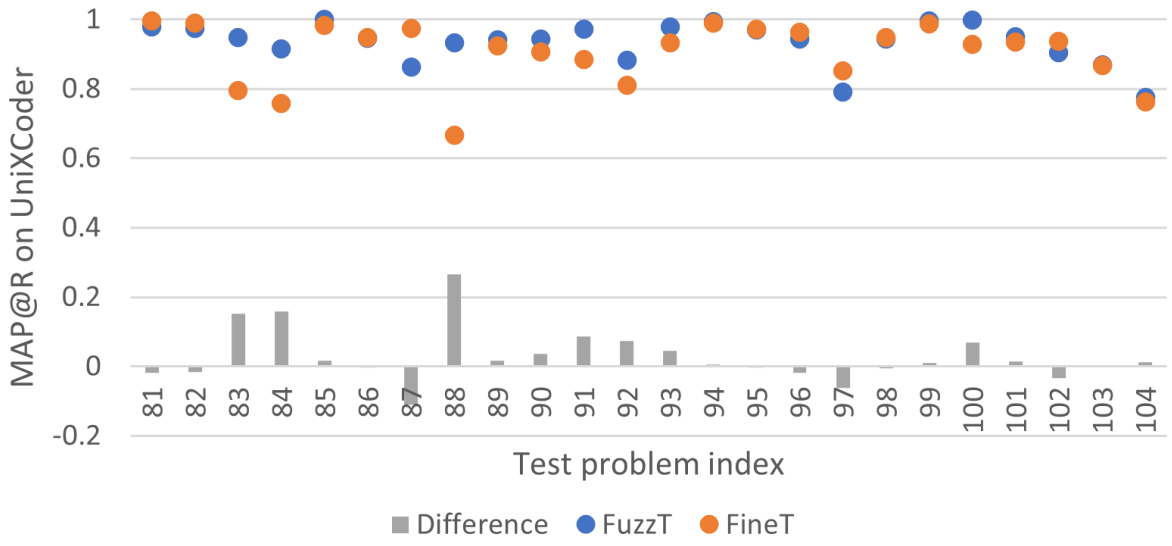


Figure 5.2: Per-problem clone detection performance on the POJ104 test set, using *UniXcoder+FineT* or *UniXcoder+FuzzT*. The horizontal axis shows the ID of the POJ104 problems, and the vertical axis is the MAP@R.

may not be the most challenging one. However, other issues similarly exist, e.g., UniXcoder+FineT shows its worst performance on Problem 88, as can be seen in Figure 5.2.

By contrast, since the programs achieve the same goal, the test cases can help convey that information to the model. This further demonstrates our idea and explains the effectiveness of our fuzz tuning.

5.5 Limitations

Fuzzers are designed to reach deep and complex control flow in large software. Many programs for current AI for code datasets do not have complex control flow. As a result, AFL++ can quickly cover all program branches before generating many inputs for us to feed to the model. We plan to try data-flow coverage as a more accurate coverage metric in the future.

AFL++ uses branch coverage to track fuzzing progress. Although it works well on C/C++

programs, it may be ineffective on languages with exceptions, which are implicit control flow. For example, AFL++ cannot distinguish different exceptions thrown in the same block, which sometimes leads to low coverage in Python programs. To overcome this issue, one possible way is to change from branch coverage to line coverage.

Although our current implementation requires a fuzzer, our approach can also work on tasks with only functions or code snippets as long as we can acquire adequate input/output pairs of the functions or code snippets, which may have some engineering challenges but is not infeasible. For example, in recent years, the software engineering community has proposed various ways to fuzz bare functions [86, 183].

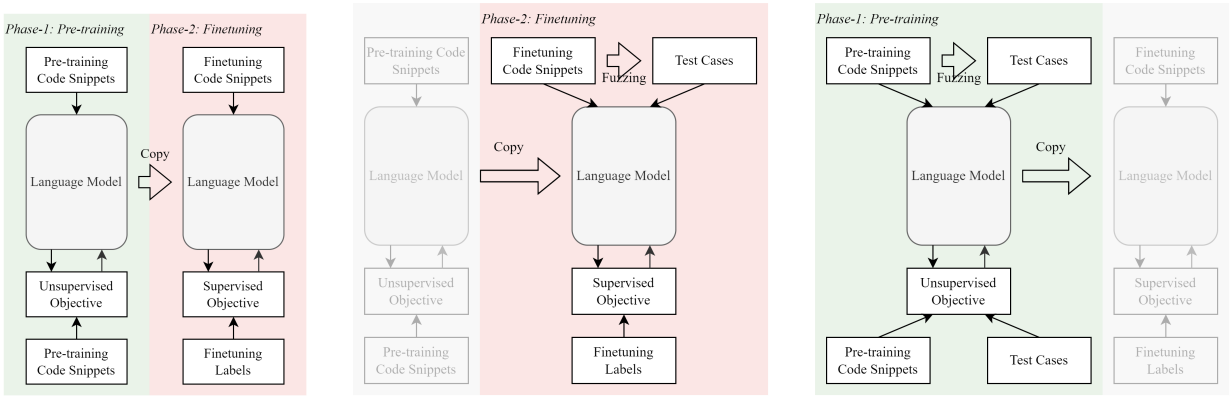
Chapter 6

Code representation pre-training with complements from fuzzer generated test cases

6.1 Introduction

Code representation learning is drawing growing attention across the community of artificial intelligence (AI) and software engineering (SE) [114, 184, 185]. It aims to abstract the structure and the underlying functionality of the source code and embed such semantics into a latent representational space via unsupervised pre-training. By providing general understanding of programs, it is a fundamental task to achieve code intelligence, which enables automated code analysis and processing by fine-tuning with budgeted computation resources or limited human annotations [168, 184, 186]. Further, as pre-trained models of code are better structured commonsense reasoners than that of natural language [187], code understanding helps with learning and reasoning for NLP related tasks.

The pre-training recipes [164, 165] for natural languages have been shown effective in code representation learning [160, 188]. However, they neglect that the structure of the code (exhibited by



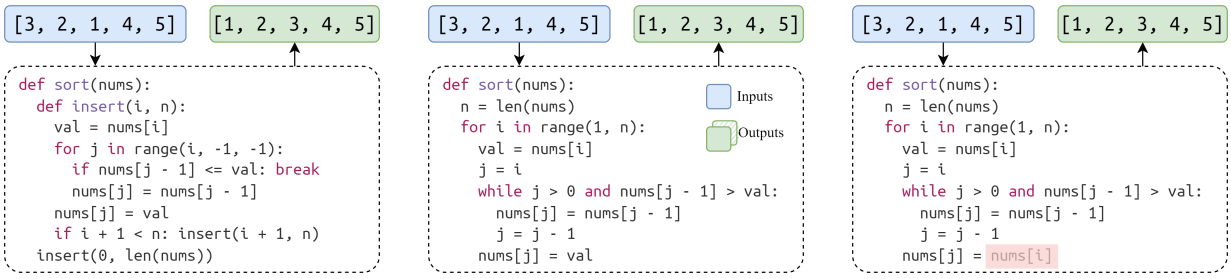
a Conventional paradigm learns to understand source code from a large scale of unlabeled code snippets, then applied the pre-trained model on downstream tasks for supervised fine-tuning by task-specific labels.

b Auto-generated test cases have been shown beneficial to code understanding on downstream tasks [116]. However, assuming the availability of test cases on every downstream task restricts their applicability by incuring additional efforts and times for program executions.

c We propose FuzzPretrain to utilize dynamic program information to aid in a more comprehensive understanding of code during pre-training. Program executions are not conducted for code on downstream tasks, hence, there is no compromise on flexibility.

Figure 6.1: An illustration of different pre-training and fine-tuning paradigms for code understanding.

how its elements, *e.g.*, variables and statements, are organized) must comply with precise and rigorous rules. These rules, often referred to as code syntax, are defined by language specification to ensure successful compilation and execution. This inspires recent works to leverage static analysis [189] to parse and present the structure of code less ambiguously by its syntactic representations, *e.g.* abstract syntax tree (AST) [169, 170] and control flow graph (CFG) [190]. However, since learning from code as it is a type of static data, existing methods overlook the underlying executable programs. Whilst programs are built with specific purposes of performing a set of behaviors indicated by their functionality, it is challenging to derive such semantics from code structure [185]. This is because the same purposes can be implemented by different algorithms in different ways, while the behaviors of programs are susceptible to subtle discrepancies in code. As depicted in Figure 6.2a and Figure 6.2b, the recursive and iterative insertion sort are notably different regarding their structures, even though they share the same functionality. Meanwhile, the subtle change in Figure 6.2c that is hard to be noticed can lead to distinct execution results. How to learn discriminative feature



a Recursive insertion sort. b Iterative insertion sort. c Buggy insertion sort.

Figure 6.2: An illustration of implementation variations of the same functional purposes. Figure 6.2a is dramatically different from its Figure 6.2b iterative counterpart regarding their structures, even though their functional equivalence is explicitly demonstrated by the consistent behaviors. On the other hand, the subtle change in Figure 6.2c is barely observable in comparisons to Figure 6.2b, but can lead to distinct execution results.

representations of code that embed not only the static information from its structure but also the dynamic information about its functionality remains an unsolved problem.

In this work, we aim to embed the functional purposes of code into its feature representations, to address the aforementioned limitations of existing works. The key idea is to abstract the behaviors of programs from their input-output relationships (represented by test cases) and enforce the model to infer such information from source code that are readily available in downstream tasks. To achieve this, we take advantage of fuzzing [143] to produce test cases that cover the logic paths of code as comprehensive as possible. Moreover, we propose a novel method called **FuzzPretrain** for joint static and dynamic information modeling. Particularly, in addition to exploring code structure by masked language modeling [165], it formulates a dynamic information matching (DIM) pretext task to tell test cases of different programs apart according to their correspondence to code. By doing so, the model learns holistic feature representations of code and its test cases, encoding both the structure and functionality. FuzzPretrain also involves a self-distillation objective to accomplish a dynamic information distillation (DID) objective. Thereby, the dynamic information is not only properly modelled but distilled from the holistic representations to code features, so to benefit in practice where the test cases are not required.

We make three contributions in this chapter: **(1)** We propose to leverage the test cases of

programs obtained with the help of fuzzing as explicit indications of functionality to complement their code and syntactic representations. To the best of our knowledge, this is the first attempt to unveil the benefits of fuzzing to code representation pre-training. **(2)** We introduce a novel code pre-training method named FuzzPretrain. It simultaneously models the structure and functionality of code while distilling from such holistic information to represent code in its feature space. It is ready to benefit downstream tasks without extra cost on test cases generations. **(3)** Extensive experiments on four code understanding downstream tasks demonstrate the effectiveness of FuzzPretrain on complementing both source code and its syntactic representations, *e.g.* AST, by dynamic program information for learning discriminative feature representations.

6.2 Related work

Code representation learning. Large language models [163–165] have achieved unprecedented breakthrough in natural language processing in recent years [165, 188, 191]. Such successes of LLMs have been consistently transferred to code representation learning and advance code intelligence. Early works in the field devote to building large-scale code corpus [68, 184] to be trained simply as plain text at the natural language conventions [160, 168, 192]. However, the rigorous syntax of programming languages exhibit additional information about code semantics [171, 193]. In this case, recent efforts turn to code-specific designs, *e.g.*, pre-training by identifier detection and infilling [167] or parsing the structure of code by its syntax [170, 190]. Despite being effective, existing approaches consider code as a type of static data and ignore the fact that it is corresponding to an executable program with unique functionality exhibited by its runtime behaviors. Such dynamic information of programs is a critical indicator to tell them apart from others with different purposes but challenging to be inferred from code structures.

Static and dynamic code analysis. Static and dynamic analysis are two common strategies in SE for ensuring the security of software products, and the former has been widely adopted in code representation learning for constructing syntactic representations [170, 171]. For early detection

of potential vulnerabilities, static analysis [189] is usually carried out by parsing the structure information of the code for inspections, while dynamic analysis [194] detects runtime errors during program executions. Fuzz testing [143], or fuzzing, is a popular dynamic analysis technique, which aims to generate a set of inputs that achieve high code coverage then feed those inputs to programs for execution to observe any unexpected outcome. Overall, the two types of code analysis strategies are usually adopted together as mutual complements to ensure the soundness and completeness of testing.

Large language models meet software testing. There are recent efforts on automated bugs mining by LLMs [195, 196], which hold an opposite objective to ours on benefiting software testing by code generation. On the other hand, harnessing program execution traces for comprehensive code representation learning has been widely studied both before and after the emergence of LLMs [185, 197–200]. Considering that fuzzing is a language-agnostic process, test cases are much easier to collect than execution traces [201] and this is crucial for constructing code understanding models with multilingual support.

Whilst [201, 202] explore the benefits of test cases for program synthesis rather than code semantically discrimination that investigated in our work, [116] share the same insight with us to leverage auto-generated test cases to benefit discriminative code understanding. However, it is noteworthy that although both [116] and us adopt a pre-training followed by a fine-tuning scheme (Figure 6.1a), there are key differences in our methodologies with a different focus on the two phases. [116] assumes the availability of test cases on every downstream tasks (Figure 6.1b), however, collecting additional information about structure or functionality of code requires sufficient expertise in SE and this undoubtedly hampers model's applicability and flexibility. By contrast, we aim to explore program executions only in pre-training to preserve the benefits of dynamic information to code understanding in practice where test cases are not mandatory (Figure 6.1c). We demonstrate in our experiments that exploring test cases only in pre-training is neither straightforward nor trivial, considering the distribution discrepancy between test cases and source code.

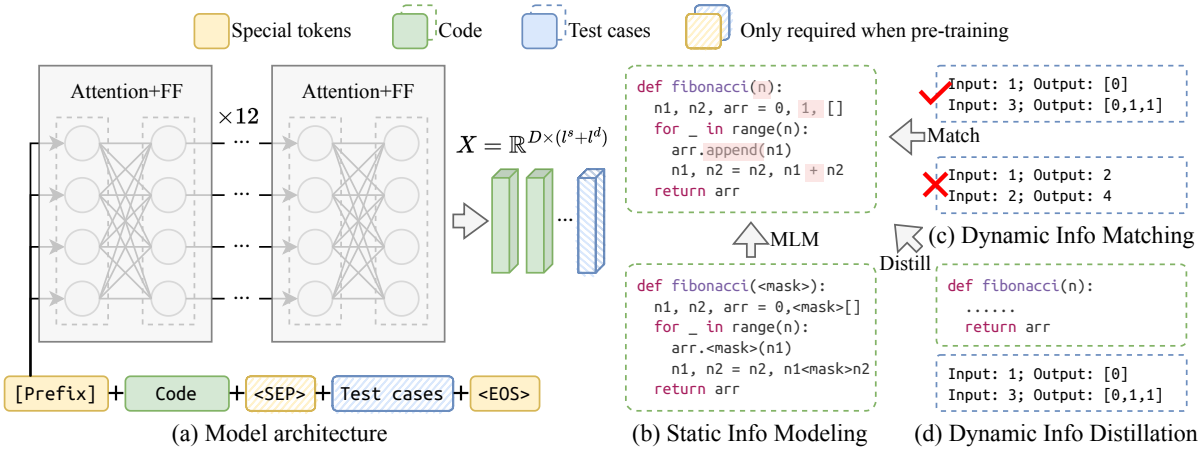


Figure 6.3: An overview of FuzzPretrain. **(a)** The input sequences are composed of both the code and test cases which are concatenated then encoded by a transformer. FuzzPretrain learns code feature representations by **(b)** static information modeling (SIM) through masked tokens predictions, **(c)** dynamic information matching (DIM) to match test cases to code, and **(d)** dynamic information distillation (DID) to summarize the holistic information about code structure and functionality.

6.3 Code representation pre-training

Given a piece of source code S and a sequence encoder f_θ parameterized by θ, our objective is to explore the underlying semantics of the code and encode them in a latent representational space $X^s = f_\theta(S) = \{\boldsymbol{x}_1^s, \boldsymbol{x}_2^s, \cdots, \boldsymbol{x}_{|S|}^s\} \in \mathbb{R}^{k\times|S|}$ in k-dimensions. This is to provide general understanding of code, which enables efficient fine-tuning on downstream tasks.

In this chapter, we propose to explore the dynamic information obtained during program executions to complement the static information learned from code structure, such that we can embed both in feature representations of code. To that end, we formulate **FuzzPretrain** whose overview is depicted in Figure 6.3. We first collect a large-scale code corpus based on CodeNet [68] and pair each code snippet with multiple test cases synthesized with the assistance of a customized fuzzer. We denote the test cases corresponding to S as D and concatenate it with the code as its joint static and dynamic representation $H = S \oplus D$. By feeding S (or H) into f_θ, the features X^s (or X^h) are trained by masked tokens predictions (Figure 6.3 (b)) and test cases to code matching (Figure 6.3 (c)). Besides, FuzzPretrain distills from the holistic features X^h of code and test cases

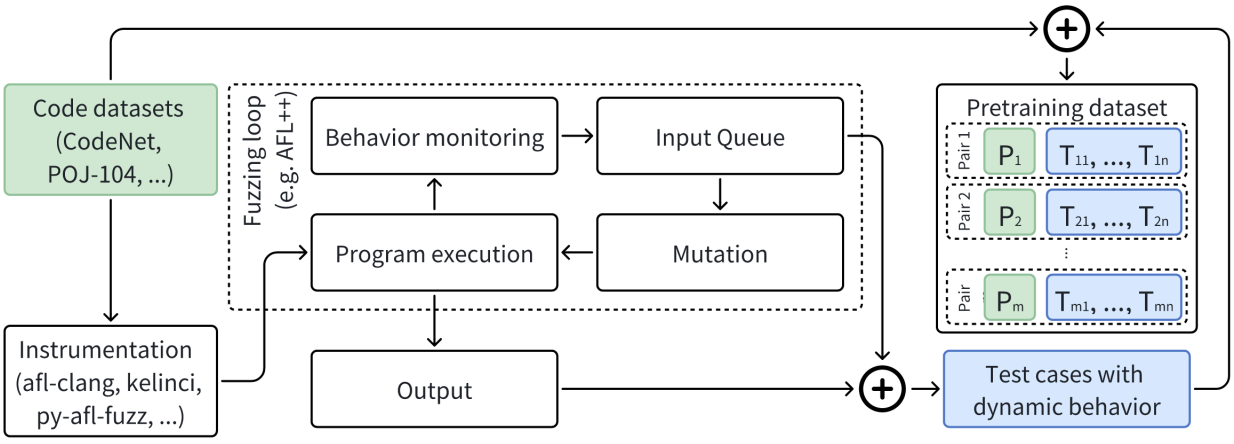


Figure 6.4: The fuzzing process collects test cases that embed dynamic behavior from program datasets.

and embed it into X^s, in order to adapt to downstream tasks where test cases D are not available.

6.3.1 Fuzzing code corpus

Fuzzing is a software verification technique that plays an important role in identifying vulnerabilities and enhancing software reliability. A fuzzer verifies the software by repeatedly generating inputs for the software to execute. For each execution, the fuzzer monitors the internal state of the software to determine if the input triggers new behavior. These inputs will be stored for future input generation. Input generation and behavior monitoring together allow the fuzzer to effectively focus on exploring new program behaviors. We believe these inputs contain runtime information that cannot be easily found using static analysis. Therefore, using those test cases, *i.e.*, program inputs and corresponding outputs, should supply extra dynamic information to the language model. We employ methods outlined in FuzzTuning [116] to carry out preprocessing, compilation, and fuzzing of the code corpus. For C++ programs, we applied the same preprocessing as in FuzzTuning and used AFL++'s [53] to perform instrumentation and compilation for fuzzing. For Java programs, we compiled them to JVM bytecode using `openjdk-18` [1] and instrument the bytecode using Kelinci

[1] `https://jdk.java.net/18/`

[174]. For Python, we modified the interpreter to report the program behaviors during execution via `py-afl-fuzz` [2]. We fuzz the program using AFL++ [53] and extract the inputs. Finally, we re-run the program and record the output of the execution. Figure 6.4 shows our fuzzing process for obtaining dynamic information from program datasets. More details about fuzzing existing code corpus can be found in [116].

6.3.2 Static and dynamic information modeling

To investigate the versatility of learning with dynamic program information, we build the FuzzPretrain upon two representative models trained on either code or AST, namely CodeBERT [160] and UniXcoder [170], respectively. We want to emphasize that FuzzPretrain is a generic method that can be integrated into arbitrary static-based models more than the two studied here. FuzzPretrain is ready to benefit different representations of code in a plug-in manner once they are serialized as a sequence of tokens. For clarity, we introduce our designs in terms of source code inputs in this section. The designs for other models like UniXcoder are similar.

Input/Output representations. As illustrated in Figure 6.3 (a), we follow [160] to concatenate different parts of inputs together with an `<SEP>` token and put an end-of-sentence `<EOS>` token to the end of the concatenation. CodeBERT adopts a begin-of-sentence token `<BOS>` as the prefix for the input sequences while UniXcoder takes `<BOS><ENCODER-ONLY><SEP>` [170]. For the test cases, we follow [116] to decode them from a series of bytes to Unicode strings and then prompt them in a form of natural language to be "Input is: `INPUT`; Output is: `OUTPUT`". This is inspired by how programming online judgement tools present problem descriptions along with its example test cases to human. We concatenate multiple test cases of a program with the `<SEP>` token to form D. We learn from multiple test cases at a time because a single test case is likely to invoke only a part of a program, and only with sufficient number of test cases can we profile the behaviors of the program comprehensively.

We follow the common recipe of natural language processing to split the concatenation of prefix,

[2] `https://pypi.org/project/python-afl/`

code, test cases, and suffix as WordPiece [203]. By feeding the token sequence S into the encoder f_θ, CodeBERT adopts the feature of the `<BOS>` token as its sequence-level representation $\boldsymbol{x}^s$ while UniXcoder applies average pooling on all the tokens to obtain $\boldsymbol{x}^s$.

Static information modeling. To learn from the structure of code S according to the dependencies among tokens, we adopt the conventional masked language modeling (MLM) which has been shown simple yet effective on context understanding [165]. We follow the common practices to randomly choose 15% of the tokens in S and replace 80% of the selections with a special `<MASK>` token, 10% with random tokens and the remaining are left unchanged. Formally, given the code S, a subset $M \subset S$ of it is masked out and leaving a sequence $\tilde{S}$ with replaced tokens. Then, the learning objective is to predict M given the context in $\tilde{S}$:

$$\mathcal{L}_{\text{SIM}}(S) = -\sum_{m \in M} \log p(m|\tilde{X}^s), \tag{6.1}$$

where m is one of the masked token and $\tilde{X}^s$ is the features of $\tilde{S}$ produced by f_θ. The term $p(m|\tilde{X}^s)$ denotes the probability that m is correctly reconstructed given the incomplete context $\tilde{X}^s$.

Dynamic information modeling. To learn from the dynamic program information obtained during executions, we propose to match the input-output mappings derived from test cases with the functionality inferred from the code. Given a code sequence S, we randomly sample an unmatched test cases list D^- and decide whether to concatenate S with its own test cases D or the negative one D^- to form an input sequence H at each training step. We then pair H with a binary label $y \in \{0, 1\}$ indicating whether the mapping relationships embedded in it are consistent. After that, H is encoded by f_θ to compute its sequence-level representation $\boldsymbol{x}^h$, which is further fed into an additional linear projection layer FC followed by a binary classifier f_ϕ:

$$\mathcal{L}_{\text{DIM}}(S, D) = \text{BCE}(y, f_\phi(\text{FC}(\boldsymbol{x}^h))). \tag{6.2}$$

In Equation 6.2, the feature $\boldsymbol{x}^h$ of H is linearly transformed the fed into the classifier f_ϕ to predict

how likely the code and test cases in H are matched. Our DIM objective is formulated as a binary cross-entropy loss (BCE) to supervised the predictions from f_ϕ by the binary label y.

With the supervision of $\mathcal{L}_{\text{DIM}}$, the model is able to derive dynamic information about the execution behaviors of programs and code, given their test cases. However, test cases are not available in many downstream tasks in practice. Therefore, we further devise a dynamic information distillation (DID) objective to simultaneously learn the holistic information from both code and test cases $H = S \oplus D$ and enforce encoding such information in the features of code S. Inspired by [204], we formulate DID in the contrastive learning paradigm to identify the holistic representation H from a list of random samples H^- according to the corresponding source code S. To be concrete, we follow [205] to maintain a stale copy $f_{\hat{\theta}}$ of the backbone encoder, which shares the identical architecture with f_θ and is updated accordingly by exponential moving average (EMA) [206]. We then compute the sequence-level feature representation $\boldsymbol{x}^s$ of S and $\hat{\boldsymbol{x}}^h$ of H by f_θ and $f_{\hat{\theta}}$, respectively. Given the holistic features X^- of a set of random samples H^- computed by $f_{\hat{\theta}}$, which are likely with different semantics from H, we train f_θ to optimize:

$$\mathcal{L}_{\text{DID}}(S, S \oplus D) = -\log \frac{g(\hat{\boldsymbol{x}}^h, \boldsymbol{x}^s)}{g(\hat{\boldsymbol{x}}^h, \boldsymbol{x}^s) + \sum_{\boldsymbol{x}^- \in X^-} g(\boldsymbol{x}^-, \boldsymbol{x}^s)}. \tag{6.3}$$

The function $g(x, y) = \exp(\cos(x, y)/\tau)$ in Equation 6.3 computes the exponential cosine similarity between two vectors where τ is a temperature hyperparameter controlling the concentration degree of the similarity distribution. In contrast to $\mathcal{L}_{\text{DIM}}$, we always compute the holistic feature $\hat{\boldsymbol{x}}^h$ of code and its own (matching) test cases to avoid the distractions from inconsistent structure and functionality.

6.3.3 Model training and inference

The proposed FuzzPretrain model is optimized alternatively according to static information modeling $\mathcal{L}_{\text{SIM}}$ (Equation 6.1), dynamic information matching $\mathcal{L}_{\text{DIM}}$ (Equation 6.2) and dynamic information distillation $\mathcal{L}_{\text{DID}}$ (Equation 6.3) on each mini-batch of data following [170, 207]. At

each training step, the stale encoder $f_{\hat{\theta}}$ is updated according to f_θ by EMA: $\hat{\theta} = \lambda\hat{\theta} + (1-\lambda)\theta$ with a momentum factor λ, and the holistic representations $\hat{\boldsymbol{x}}^h$ obtained from code and its corresponding test cases will be fed into the queue X^- with the oldest ones inside being removed in a first-in-first-out manner. After pre-training, we keep only the transformer encoder f_θ which is able to yield discriminative feature representations of code $X^s = f_\theta(S)$ when only it is available but not the test cases D at inference or on downstream tasks.

6.4 Experiments

Datasets. [68] proposed a large-scale dataset CodeNet, consisting of over 14 million code samples and about 500 million lines of code, which is intended for training and evaluating code models. We adopted the C++1000, C++1400, Python800, Java250 benchmark datasets of CodeNet[3] to be fuzzed as the training data of FuzzPretrain. We then evaluated FuzzPretrain extensively on four code understanding benchmark datasets of CodeXGLUE [168][4]: (1) another subset of **CodeNet** [68] collected by [170] consisting of 50K functions implemented in Python, Java, and Ruby for solving one of $4,053$ online coding problems; (2) **POJ-104** [67] which contains 104 C/C++ coding problems with 500 code submissions to each; (3) **Devign** [208] which is composed of vulnerable functions from four large and popular C-language open-source projects with manual labels; (4) **CosQA** which contains 20,604 pairs of code and real-world web queries [209] with annotations from human experts indicating whether the questions raised by the queries can be properly addressed by the code. All the data used for fine-tuning and testing are carefully aligned with previous studies [160, 170].

Evaluation protocols. We first investigated the discrimination ability of the learned code representations by code-to-code search (abbreviated as *code search* in the chapter) on the subset of CodeNet collected by [170]. In this task, submissions of the same coding problems are assumed to share the same semantics regardless of their implementations. The feature distances between code pairs were adopted to measure their semantic similarity and the mean average precision (**mAP**) was

[3]Licensed under the Apache License, Version 2.0
[4]Licensed under Creative Commons Zero v1.0 Universal

reported to quantify the quality of the retrieval results. We then studied the effects of FuzzPretrain to several downstream tasks in unseen domains, including clone detection, defect detection and text-to-code search (abbreviated as *text search*). The objective of clone detection is similar to that of code search but with fine-tuning in target domains. We followed the same protocol of [160]'s work to test on POJ-104 and use **mAP**@R to assess the results, with only the top-R ($R = 499$) most similar samples were considered in retrieval. In the task of text search, which requires retrieving code snippets according to textual queries, the mean reciprocal rank (**MRR**) is adopted as the metric following [170]'s work. This evaluation was conducted on CosQA. Defect detection was carried out on Devign and the accuracy (**Acc**) of binary classification is adopted with a fixed threshold of 0.5.

Implementation details. Both our base models, *i.e.* CodeBERT [160] and UniXcoder [170], followed [164] to take a 12-layers transformer with 125M learnable parameters for sequence encoding. We followed their designs to set the batch size to 2048 and 1024 while the maximum sequence length to 512 and 1024 for CodeBERT and UniXcoder, respectively. In inputs, 400 and 800 tokens are reserved for code and AST, respectively, and the rest are for test cases. The test cases of each program were concatenated with the code or the AST by the separation token until reaching the length limits, while the rest was dropped. The FuzzPretrain model was updated by the Adam optimizer [182] during training with a learning rate of $2e-5$ for $10K$ steps. For dynamic information distillation $\mathcal{L}_{\text{DID}}$ (Equation 6.3), we followed [205] to set the momentum coefficient $m = 0.999$, the temperature $\tau = 0.07$, and the number of random samples $|H^-| = 2^{16}$. The overall pre-training process took round 12/20 hours on 8 Nvidia V100 GPUs for training with code and AST, respectively.

6.4.1 Code representation learning

Learning with modality discrepancy. To study whether the inconsistency between pre-training and deployment will refrain FuzzPretrain from benefiting general code understanding, we adopted the code search task to identify equivalent functions without fine-tuning the code representations or learning additional classifiers. Considering that FuzzPretrain was trained on different data from its base models (CodeBERT and UniXcoder), to derive reliable conclusions

Table 6.1: Evaluations on code search. Results of our base models (CodeBERT and UniXcoder) are from [170]'s paper, which are marked in grey because of different training data. The first and second rows in the header indicate the programming language of the query and the target code snippets, respectively. The column "DYN" indicates whether a model was trained using the test cases or not. mAP scores (%) are reported.

Model	DYN	Ruby			Python			Java			Overall
		Ruby	Python	Java	Ruby	Python	Java	Ruby	Python	Java	
CodeBERT	✗	13.55	3.18	0.71	3.12	14.39	0.96	0.55	0.42	7.62	4.94
CodeBERT-MLM	✗	22.45	5.67	1.95	6.74	25.70	5.01	3.61	5.84	13.45	10.05
CodeBERT-MLM+RTD	✗	13.22	1.00	0.10	1.24	14.35	1.20	0.20	0.18	6.34	4.20
FuzzCodeBERT	✓	**27.92**	**14.88**	**7.92**	15.39	**30.47**	**10.26**	**9.94**	**10.65**	**17.75**	**16.13**
FuzzCodeBERT w/o DIM	✓	24.05	14.08	6.96	**16.32**	27.51	9.54	8.66	9.76	13.49	14.49
FuzzCodeBERT w/o DID	✓	18.21	2.92	0.72	2.88	25.67	3.13	0.80	1.98	17.98	8.25
UniXcoder	✗	29.05	26.36	15.16	23.96	30.15	15.07	13.61	14.53	16.12	20.45
UniXcoder-MLM	✗	20.49	13.54	3.25	10.40	19.49	3.69	4.13	5.14	12.29	10.27
UniXcoder-MLM+Contrast	✗	30.83	25.73	16.46	25.44	30.50	16.80	16.01	17.26	18.86	21.99
FuzzUniXcoder	✓	**42.84**	**29.83**	**17.70**	**33.73**	**47.77**	**21.94**	**20.83**	**23.52**	**33.78**	**30.22**
FuzzUniXcoder w/o DIM	✓	22.50	13.52	6.66	15.31	22.99	6.81	7.54	6.84	12.94	12.79
FuzzUniXcoder w/o DID	✓	12.92	5.10	1.36	5.56	14.86	0.87	0.96	0.50	6.81	5.44

from fair comparisons, we built several fairer baselines. The baselines were trained under the exactly same settings as FuzzPretrain but learning from only code or AST without test cases. We presented CodeBERT/UniXcoder-MLM to train by MLM solely as the baselines following [179], and CodeBERT-MLM+RTD/UniXcoder-MLM+Contrast to adopt all the losses dedicated to code understanding in their papers for comprehensive exploration on static information modeling. Correspondingly, we denote the two variants of FuzzPretrain built upon CodeBERT and UniXcoder as *FuzzCodeBERT* and *FuzzUniXcoder*, respectively.

As shown in Table 6.1, the superior performances attained by FuzzCodeBERT and FuzzUniXcoder over their static baselines demonstrate that FuzzPretrain is able to yield discriminative code representations that are beneficial to downstream tasks where test cases are not given. We attribute the performance superiority obtained by FuzzPretrain to the designs of not only modeling the dynamic information jointly from code and test cases but also distilling such knowledge to be encoded into the feature representations of code. This is evident by the degradation of FuzzPretrain when training without either of the proposed components. Such performance drops further verify the effectiveness of our delicate designs and demonstrate that it is non-trivial to benefit code

Reference

```
def iroha():
    a, b, c = input().split()
    s = a[len(a)-1]
    sshead = b[0]
    sstail = b[len(b)-1]
    sss = c[0]
    if s == sshead and sstail == sss:
        print("YES")
    else:
        print("NO")
if __name__ == "__main__":
    iroha()
```

NN from UniXcoder

```
def iroha():
    a, b, c = input().split()
    shead = a[0].upper()
    sshead = b[0].upper()
    ssshead = c[0].upper()
    print(shead + sshead + ssshead)
if __name__ == "__main__":
    iroha()
```

NN from FuzzPretrain

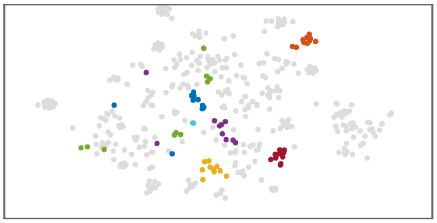
```
a,b,c = input().split()
if a[len(a)-1] == b[0] and b[len(b)-1] == c[0]:
    print("YES")
else:
    print("NO")
```

Anchor

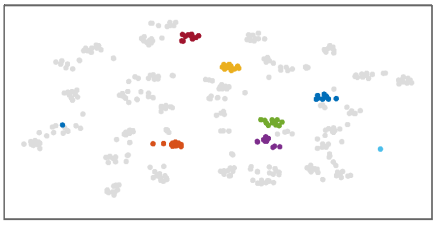
False positive

True positive

(a) A case study



(b) Features from UniXcoder



(c) Features from FuzzPretrain

Figure 6.5: Qualitative studies for code search. The functional equivalence of code snippets are marked by their shared colors. Only a few classes are highlighted with bright colors to be visually distinguishable.

representation learning by dynamic program information. It is also noteworthy that applying either DIM or DID solely benefits CodeBERT but not UniXcoder. Our hypothesis is that there is a trade-off between the new knowledge acquired and the prior knowledge forgotten during continual learning. It is always easier for the new knowledge to outweigh the prior for a less optimal base model.

Qualitative studies. For more intuitive understanding of FuzzPretrain's advantages on code search, we show an example in Figure 6.5 (a) to exhibit the nearest neighbors of a reference code snippet decided by either UniXcoder or its FuzzPretrain counterpart. It is not surprising that the code with similar structure (*e.g.* variable or function names and the main entry point) can be easily confused with each other by the static-based method even though the false positive example is with different purposes from the reference. On the other hand, such functionality-wise relationships between code snippets are exposed by their execution behaviors, hence, are well captured by FuzzPretrain. To provide a global picture of the learned code features, we adopted t-SNE [210] to visualize the python code for 50 randomly selected problems (classes), which were

Table 6.2: Evaluations of code representations on inductive code search.

	c1000	c1400	py800	java250	Overall
CodeBERT	13.95	13.22	31.23	26.72	21.28
CodeBERT+MLM	26.34	24.08	48.71	34.94	33.52
FuzzCodeBERT	69.98	68.65	78.13	69.98	71.69
UniXcoder	17.57	15.89	55.28	45.49	33.56
UniXcoder+MLM	32.84	30.28	46.79	46.90	39.20
FuzzUniXcoder	71.72	68.40	80.27	77.43	74.45

encoded by the static-based model or our FuzzPretrain in Figure 6.5 (b) and (c), respectively. The functional equivalence of code are highlighted by the same colors, with only a few random problems are marked with bright colors to avoid chaos. As depicted, the features of functionally equivalent code snippets yielded by UniXcoder can sometimes spread over the feature space sparsely due to their implementation variations. However, our FuzzPretrain forms more compact clusters which are consistent with the underlying semantics of code. These visualizations help explain the potential benefits of jointly learning from the static and dynamic information to comprehensive code understanding.

Inductive zero-shot code search. We adopted the testing split provided by UniXcoder [170] for evaluation of code search, it is likely to overlap with our training data in CodeNet by sharing the coding problems. Therefore, we consider the searching of those overlapping samples as transductive inference problems. This is also a practical scenario given that the training data of the latest code models covers a large proportion of open-source projects in Github and is likely to involve the code-of-interests to users. We have also evaluated in an inductive setup where the query and the candidate code snippets are submissions to 50 coding problems of each programming language that have never been seen during pre-training. As shown in Table 6.2, the superiority of our FuzzPretrain over both the base models and our baselines still holds. That is, these results show that our model is effective not only in the transductive inference setup for code search, but also in an inductive setup where no training/test overlap exist.

Table 6.3: Evaluations in novel data domains. Results of the base models are marked in grey as training on different data from ours. Results marked with * are reproduced using the checkpoints from authors.

Model	DYN	Clone	Defect	Text
CodeBERT	✗	82.7	62.1	65.7
CodeBERT-MLM	✗	88.7	63.5	67.4
CodeBERT-MLM+RTD	✗	84.7	62.0	66.3
FuzzCodeBERT	✓	**93.0**	**64.1**	**69.1**
UniXcoder	✗	90.5	64.5*	70.1
UniXcoder-MLM	✗	91.2	63.8	69.8
UniXcoder-MLM+Contrast	✗	91.1	**65.2**	69.7
FuzzUniXcoder	✓	**92.2**	64.5	**70.7**

Code understanding in novel domains. We investigated whether our learned code features are transferable and beneficial to downstream tasks in unseen data domains [168] in Table 6.3. We see non-negligible performance advantages obtained by our FuzzPretrain over these static-based methods which learn from only the code structures. Although introducing contrastive learning by feeding the same code inputs to the encoder twice [211] (*i.e.*, "Contrast" in Table 6.3) is helpful to UniXcoder-MLM on defect detection, it leads to subtle performance degradation on the other two tasks. In fact, FuzzPretrain can obtain a similar improvement (from 64.5% to **65.6%**) by integrating such a code-to-code contrast as well. This implies the potential of our dynamic information modeling on more advanced base models.

Comparisons with more state-of-the-arts. Although FuzzPretrain adopted different pre-training data from the popular bi-modal dataset [184] to enable compilation and fuzzing, we compared it with the state-of-the-art models regardless to demonstrate its competitiveness on code understanding. Specifically, we compared FuzzPretrain with three types of methods. RoBERTa [164] learns at the natural language conventions. DISCO [212], CodeRetriever [213], and ContraBERT [179] benefit from contrastive learning as in our solution. GraphCodeBERT [171], CodeExecutor [185] and TRACED [200] explore program functionality from DFG or execution traces. Note that, we evaluated CodeExecutor without re-ranking by execution traces to be more practical.

Table 6.4: Comparisons with the state-of-the-arts that adopt the same backbone network as ours with 125M parameters. Results marked with * are reproduced using the checkpoints from authors.

Model (Year)	Clone	Defect	Text
RoBERTa (2019)	76.7	61.0	60.3
GraphCodeBERT (2020)	85.2	62.9	68.4
DISCO (2022)	82.8	63.8	-
CodeRetriever (2022)	88.8	-	69.7
ContraBERT (2023)	90.5	64.2	66.7*
CodeExecutor (2023)	70.5*	59.0*	13.1*
TRACED (2024)	91.2	**65.9**	-
FuzzCodeBERT	**93.0**	64.1	69.1
FuzzUniXcoder	92.2	64.5	**70.7**

As shown in Table 6.4, the performance advantages of FuzzPretrain over GraphCodeBERT implies that mining the functionality of programs from the intricate dependencies among variables is more challenging than modeling from the concrete input-output behavior represented by test cases. Besides, TRACED is good at code understanding in finer granularity (*e.g.* defect detection) by learning from the detailed internal status of programs in execution traces while our FuzzPretrain is superior on global understanding of code snippets (*e.g.* clone detection) as the test cases we adopted is invariant to implementation variations that are agnostic to functionality. Whilst the methods that are based on contrastive learning of source code yielded promising results, FuzzPretrain's competitiveness demonstrate the effectiveness of learning with complements from dynamic information. More importantly, FuzzPretrain can be integrated into those methods to further benefit from more advanced modeling of static information.

Comparisons to commercial LLMs. Commercial LLMs have recently shown remarkable zero-shot capability to various code understanding and generation downstream tasks. Whilst our proposed ideas are generic and integrable into any static-based models regardless of their scale, we further conducted a preliminary comparison to the "text-embedding-ada-002" model from OpenAI to demonstrate our specialization on the task of semantic code search. To be concrete, we followed OpenAI's instruction of getting code embeddings[5] to evaluate their model for python-to-python

[5]https://platform.openai.com/docs/guides/embeddings/use-cases

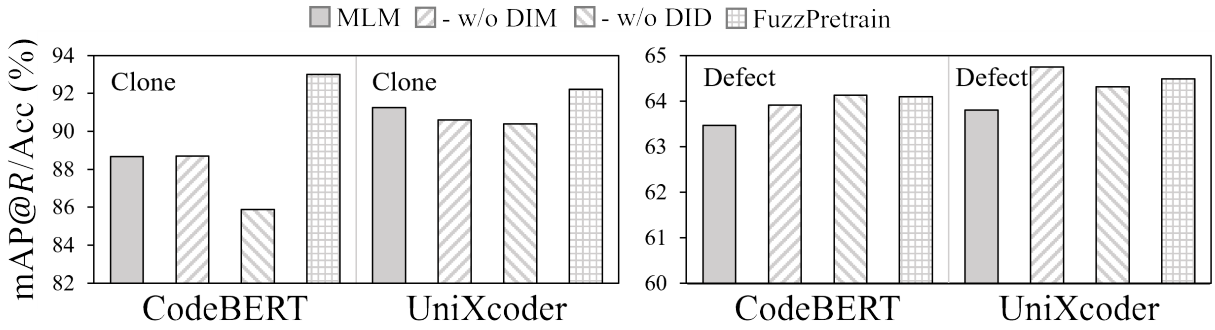


Figure 6.6: Effects of different components for dynamic information modeling. We constructed three variants of FuzzPretrain with either DIM or DID or both being removed to be compared.

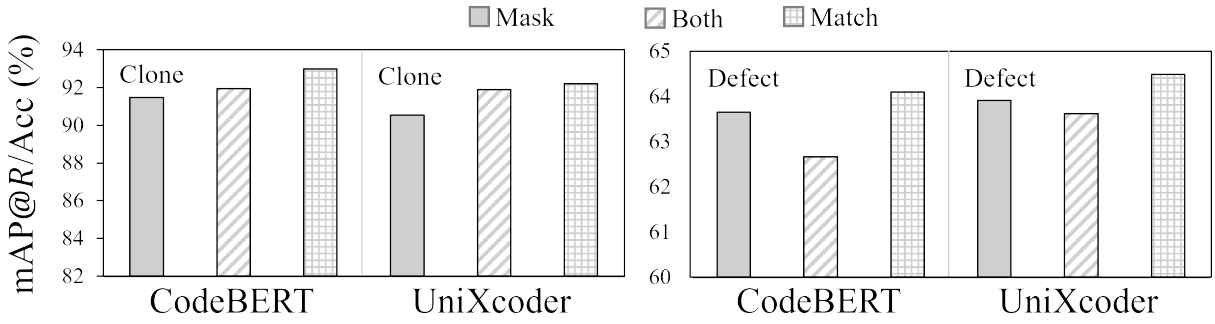


Figure 6.7: Dynamic information modeling by MLM. The "Mask" variant replaces DIM by MLM for both code and test cases while "Match" is the design we adopted and "Both" is the combination of the two.

code search in CodeNet [68]. The OpenAI's model yielded 35.91% mAP while ours are 30.47% and 47.77% when adopting either CodeBERT or UniXcoder as the base model, respectively. Given that CodeNet carefully removed near-duplicated submissions to the same coding problems with over-high syntactic similarity, such initial evaluation results indicate that semantic code search is fundamentally challenging and the test cases we adopted are strong indicators of program's functionality, which ensure our competitiveness to the larger and more complex models.

6.4.2 Ablation study

Effects of dynamic information modeling. To study the independent contributions of DIM (Equation 6.2) and DID (Equation 6.3) to dynamic information modeling, we constructed and

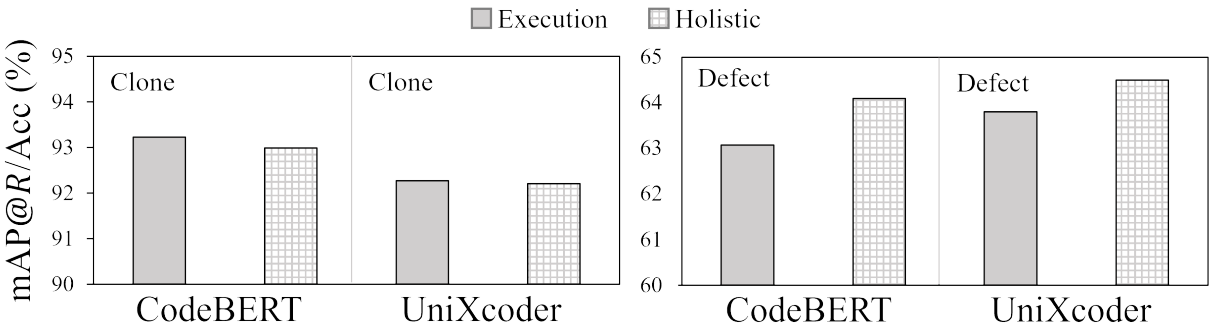


Figure 6.8: Positive pairs in DID. The "Execution" variant constructs the positive pairs in DID using code T^s and its test cases T^d, and our "Holistic" design contrasts code to its concatenation with test cases $T^s \oplus T^d$.

compared three variants of FuzzPretrain by removing either or both of them. As shown in Figure 6.6, the variant of FuzzPretrain trained with only DID (w/o DIM) often out-performed the baselines (MLM) trained with neither DIM nor DID. This indicates that the test cases concatenated after the source code or its syntactic representations potentially play the roles of data augmentation to perturb the distributions of code by supplementing the dynamic information from test cases. Although adopting either DIM or DID is slightly better than FuzzPretrain occasionally, the consistent improvements we brought to different base models on both the retrieval (clone detection) and classification (defect detection) tasks demonstrate the generality of combining the two designs, which is critical for a pre-training method.

Dynamic information modeling using MLM. To justify our DIM's effectiveness on dynamic modelling over the conventional MLM, we replace or combine it with MLM on both code and test cases to form two variants of FuzzPretrain as "Mask" and "Both" in Figure 6.7, respectively. The performance superiority of "Match" to the two variants indicates that applying MLM in test cases is sub-optimal. From our training logs, we observe that the encoder could accurately reconstruct the masked tokens in test cases (or code) regardless of whether the code (or the test cases) is available in the model input. This implies that syntactic and functional representations are both very informative and can be well reconstructed independently, which makes it less straightforward to associate them by MLM. On the contrary, the labels for our DIM is defined only by the relationships between

code and test cases, hence, it is infeasible to predict such labels without learning their correlations. Besides, the "Both" alternative tends to associate code with arbitrary patterns in test cases, which are explored by MLM. The resulted correlations can be distracting to code understanding considering the randomness in test cases introduced by fuzzing.

Positive pairs in DID. To justify our design of DID, we built a variant of FuzzPretrain which formulates the $\mathcal{L}_{\text{DID}}$ to identify test cases D according to their corresponding code S or AST by constructing the positive pairs in Equation 6.3 to be (S, D) instead of $(S, S \oplus D)$ in FuzzPretrain. We denote this variant as "Execution" and FuzzPretrain as "Holistic" to be compared in Figure 6.8. Although the performances of the "Execution" variant on clone detection are on par with that of the "Holistic" counterpart, its inferiority on defect detection is non-negligible. We believe that this is due to the distribution discrepancies between code and test cases (*e.g.* test cases are likely to involve an exhaustive list of random numbers as inputs which are barely seen in code). It is more reasonable to jointly learn from test cases and source code to simultaneously benefit from dynamic information and mitigate the negative impacts from distribution discrepancies.

6.5 Future work and limitations

Fuzzing code corpus. Our current pre-training data is restricted to OJ-like code corpus (*i.e.*, CodeNet) [68], which refrains us from ablating affecting factors in the data distribution in making fair comparison to existing methods. To be more specific, most commonly adopted code corpus [184] are composed of standalone functions spread over various software projects (*e.g.*, CodeSearchNet), whose test cases cannot be easily obtained. Whilst OJ data is showing some unique characteristics to benefit our FuzzPretrain model on understanding similar code snippets as indicated by our remarkable performance advantages on POJ-104 [67] (Table 6.4), this also limits our model's generalization ability to other type of code corpus, *e.g.* the F1-score of clone detection on BigCloneBench [66] yielded by our FuzzUniXcoder was 1% lower than that by UniXcoder pre-trained on CodeSearchNet. Yet, when both pre-trained on the same selected subset of CodeNet, our FuzzPretrain leads to

+0.9% F1 gain in comparison to existing pre-training strategies using, for example, the MLM loss on CodeBERT. Exploring fuzzing on more diverse code corpus help address this limitation.

Text-code tasks. Following the discussion about fuzzing code corpus in the previous paragraph, we would like to mention that, since CodeNet does not contain text description of each code, pre-training on it may not fully unleash the power of pre-training on text-code downstream tasks. That is to say, although we have shown the effectiveness of our FuzzPretrain on the text code search task in Table 6.3 and Table 6.4, even better results can be obtained if we can pair the CodeNet data with text descriptions or if we can pre-train on a dataset with not only texts and code but also test cases. This also withholds FuzzCodeBERT and FuzzUniXcoder from surpassing every state-of-the-art methods on text-code tasks. In addition to exploiting datasets, extensive experiments presented in this chapter also verifies complementary effects of dynamic program modeling to these methods, which implies that combining more advanced methods [214, 215] with our FuzzPretrain also leads to superior performance than that of FuzzCodeBERT and FuzzUniXcoder.

Code generation. Our designs for dynamic information modeling are all about the holistic comprehensions of code in a global picture, while how to benefit token-wise code understanding by using it is not straightforward. We tested UniXcoder with and without our FuzzPretrain on the python dev split of the line-level code completion task in the CodeXGLUE benchmark [168], our FuzzUniXcoder yielded 42.73%/72.03% Exact Match/Edit Sim *vs*. 42.68%/71.88% by UniXcoder. We did not observe clear improvements brought by FuzzPretrain on code generation tasks which are usually conducted at token-level, leaving an interesting problem to be studied in the future.

Chapter 7

Conclusion

In this thesis, we explore three aspects of greybox fuzzing that have not been well studied. First, we study the necessary improvements in fuzzers to trigger more bugs. Then, we evaluate the effectiveness of specialized fuzzing on the LLVM backend. Finally, we discover a way to use the byproduct of fuzzing to enhance the code understanding ability of large language models (LLMs). Still, there are many aspects that can be further studied. For example, it is challenging for fuzzers to automatically adapt to different input formats [43, 44]. However, with the development of LLMs, it is worth investigating how LLMs can assist fuzzers in generating inputs that fit various formats [46]. We summarize the contributions and potential future work in this chapter.

7.1 Improving fuzzing performance using principled techniques

We first design and implement Integrity to trigger integer arithmetic errors using fuzzing in Chapter 2. By identifying and instrumenting integer arithmetic operations with potential errors, Integrity provides critical information to the fuzzer to help it trigger potential bugs. Integrity found all the integer errors in the Juliet test suite with no false positive. On nine popular open-source programs, Integrity discovered a total of 174 true errors, including eight crashes and 166 non-crashing errors. To efficiently determine whether a non-crashing error is harmful, we propose two methods to

find potentially harmful errors. The first one is based on the statistics of traces produced by the fuzzer. On the other hand, we can compare the output of independent implementations of the same algorithm on the same input. Our evaluation demonstrated that Integrity is effective in finding integer errors.

With the experience from developing Integrity, we identify the challenges that the state-of-the-art mutation-based greybox fuzzers face when finding vulnerabilities in real-world scenarios. State-of-the-art fuzzers cannot achieve better performance for two reasons. First, they lack accurate and fine-grained branch counting feedback. Additionally, their mutation strategies are not well-suited to real-world scenarios. In Chapter 3, we propose a new fuzzer, Valkyrie to address these challenges. Valkyrie implements collision-free context-sensitive branch counting, which eliminates branch collisions while preserving context sensitivity. Besides, Valkyrie implements a predicate solver for fuzzing that adapts optimization algorithms from the real domain to the integer domain. Finally, we use the solver to assist in triggering bugs by converting potentially exploitable code into predicates.

We evaluate Valkyrie on the Magma benchmark as well as real-world programs. Our results show that Valkyrie triggers 21 unique integer and memory errors, 10.5% and 50% more than AFL++ and Angora, respectively. In real-world programs, Valkyrie's branch counting mechanism proved effective by eliminating branch collisions and maintaining context-sensitivity, while AFL++ and Angora incur high bitmap utilization rates, indicating significant branch collision probabilities. For coverage statistics, Valkyrie reached 8.2% more branches on average compared with AFL++, and 12.4% compared with Angora.

7.2 Specialized fuzzing for the LLVM backend

Even though generic fuzzing has helped the community identify thousands of bugs, we argue that specialized fuzzing is necessary. We propose IRFuzzer in Chapter 4, a fuzzer specializing in fuzzing LLVM instruction selection. To generate inputs that are semantically and syntactically correct, we

first identify the challenges in IR generation that don't exist in high-level language generation. We create a mutator that maintains semantic correctness by splitting a block and inserting a sCFG in between. Then, we ensure that the IR instruction we inserted is syntactically correct using a descriptive language to model all IR instructions. Therefore, the IR program we generate can always be compiled by the backend. We also propose a new coverage metric to better monitor the fuzzing status and keep track of program behavior. IRFuzzer also decodes this coverage table for mutation guidance.

Our evaluation shows that IRFuzzer outperforms existing backend and end-to-end state-of-the-art fuzzers. IRFuzzer achieved higher matcher table coverage in all LLVM backend architectures. Specialized fuzzing is required for specialized tools, and IRFuzzer is efficient enough to be used within the context of a regular development process.

Using IRFuzzer, we have identified 78 bugs in upstream LLVM code that have been confirmed by developers. Except for six bugs that will hang the compiler, all others will crash it. 57 of these bugs have been fixed, demonstrating that these bugs provide useful insights to LLVM developers. These findings indicate that there are fertile opportunities for specialized fuzzing despite the popularity of end-to-end compiler testing. We demonstrated that IRFuzzer is effective in finding bugs in the LLVM backend. We have also open-sourced IRFuzzer [1] to assist the community in testing various downstream compilers.

7.3 Enhancing program understanding using fuzzer generated test cases

Finally, we have found a valuable use for the inputs generated by the fuzzer, which would otherwise be discarded. In Chapter 6, we point out that exploiting informative test cases helps in the understanding of programs. We have developed fuzz tuning, a novel method that takes advantage of fuzzing together with prior large-scale pre-training efforts to achieve this goal. Our fuzz tuning

[1] `https://github.com/SecurityLab-UCD/IRFuzzer`

repurposes fuzzers to generate informative test cases that well-represent the functionality of programs, and it introduces appropriate cloze prompts to incorporate the test cases into the processing. By performing comprehensive experiments on two datasets and two program understanding tasks, we have verified the effectiveness of this method and achieved new state-of-the-arts.

As a follow-up, we make the first attempt to facilitate comprehensive program profiling and effective code representation learning using the test cases. To benefit from such a new modality of data that is often not available in downstream tasks, we proposed FuzzPretrain for joint static and dynamic information modeling. Specifically, FuzzPretrain is trained not only to accomplish the conventional masked tokens prediction objective but also to learn the input-output relationships from test cases encoding the program-specific runtime behaviors, as well as enforcing the model to infer such dynamic knowledge from code structures solely. Extensive experiments were conducted on various code understanding downstream tasks. The notable performance advantages yielded by FuzzPretrain over models learned from only the structure of code demonstrate the potential benefits of the complements from program executions.

7.4 Future work

As shown in Chapter 4, a fuzzer with a customized mutator can greatly improve the fuzzing results. However, it is challenging for fuzzers with little expert knowledge to adapt to different file formats. On the other hand, LLMs demonstrate a high capability in understanding and adapting to various contexts. We hypothesize that LLMs can understand and generate various input formats given proper training. Some prior attempts have been made to use LLMs for fuzzing [216–219]. However, these approaches focus only on text-based inputs and do not scale well.

We envision a fuzzer whose mutation engine is based on an LLM to harness the power of LLMs. We plan to fine-tune an LLM so it can adapt to non-text file formats like `jpeg`, `mp4`, `elf`, etc. To achieve this, we encode each file in a hexadecimal format for the LLM to process. We have fine-tuned based on llama2-7b [220]. Our initial evaluation shows that this fuzzer can outperform

the state-of-the-art fuzzers on Magma [52]. However, further investigation is required to determine whether LLMs can help detect vulnerabilities in real-world programs.

Bibliography

[1] *American Fuzzy Lop*. Accessed: 2024-03-24. URL: http://lcamtuf.coredump.cx/afl/.

[2] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591295.

[3] Karine Even-Mendoza et al. "GrayC: Greybox Fuzzing of Compilers and Analysers for C". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1219–1231. DOI: 10.1145/3597926.3598130.

[4] Junjie Wang et al. "FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-junjie.

[5] Frédéric Tuong et al. "SymRustC: A Hybrid Fuzzer for Rust". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. ¡conf-loc¿, ¡city¿Seattle¡/city¿, ¡state¿WA¡/state¿, ¡country¿USA¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2023, pp. 1515–1518. ISBN: 9798400702211. DOI: 10.1145/3597926.3604927.

[6] Sven Smolka et al. “Fuzz on the Beach: Fuzzing Solana Smart Contracts”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1197–1211. ISBN: 9798400700507. DOI: 10.1145/3576915.3623178.

[7] Jianzhong Su et al. “Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3560429.

[8] Chenyang Lyu et al. “MINER: a hybrid data-driven approach for REST API fuzzing”. In: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC '23. Anaheim, CA, USA: USENIX Association, 2023. ISBN: 978-1-939133-37-3.

[9] Yi Liu et al. “Morest: model-based RESTful API testing with execution feedback”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1406–1417. ISBN: 9781450392211. DOI: 10.1145/3510003.3510133.

[10] Peng Chen et al. “Hopper: Interpretative Fuzzing for Libraries”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1600–1614. ISBN: 9798400700507. DOI: 10.1145/3576915.3616610.

[11] Andrea Fioraldi et al. “LibAFL: A Framework to Build Modular and Reusable Fuzzers”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1051–1065. ISBN: 9781450394505. DOI: 10.1145/3548606.3560602.

[12] Harrison Green and Thanassis Avgerinos. “GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs”. In: *Proceedings of the 44th International Conference on Software*

Engineering. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1070–1081. ISBN: 9781450392211. DOI: 10.1145/3510003.3510228.

[13] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. "RULF: Rust Library Fuzzing via API Dependency Graph Traversal". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 581–592. DOI: 10.1109/ASE51524.2021.9678813.

[14] Sergej Schumilo et al. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[15] Dae R. Jeong et al. "Razzer: Finding Kernel Race Bugs through Fuzzing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 754–768. DOI: 10.1109/SP.2019.00017.

[16] Meng Xu et al. "Krace: Data Race Fuzzing for Kernel File Systems". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1643–1660. DOI: 10.1109/SP40000.2020.00078.

[17] Wen Xu et al. "Fuzzing File Systems via Two-Dimensional Input Space Exploration". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 818–834. DOI: 10.1109/SP.2019.00035.

[18] Dae R. Jeong et al. "SegFuzz: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2104–2121. DOI: 10.1109/SP46215.2023.10179398.

[19] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing." In: *NDSS*. 2018.

[20] Xueqiang Wang et al. "Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1151–1167. ISBN: 978-1-939133-06-9.

URL: https://www.usenix.org/conference/usenixsecurity19/presentation/wang-xueqiang.

[21] Yaowen Zheng et al. "Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 417–428. ISBN: 9781450393799. DOI: 10.1145/3533767.3534414.

[22] Timothy Trippel et al. "Fuzzing Hardware Like Software". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel.

[23] Danning Xie et al. "DocTer: documentation-guided fuzzing for testing deep learning API functions". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 176–188. ISBN: 9781450393799. DOI: 10.1145/3533767.3534220.

[24] Yinlin Deng et al. "Fuzzing deep-learning libraries via automated relational API inference". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 44–56. ISBN: 9781450394130. DOI: 10.1145/3540250.3549085.

[25] Anjiang Wei et al. "Free lunch for testing: fuzzing deep-learning libraries from open source". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 995–1007. ISBN: 9781450392211. DOI: 10.1145/3510003.3510041.

[26] Jiazhen Gu et al. "Muffin: testing deep learning libraries via neural architecture fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22.

Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1418–1430. ISBN: 9781450392211. DOI: 10.1145/3510003.3510092.

[27] Dongdong She et al. "NEUZZ: Efficient Fuzzing with Neural Program Smoothing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 803–817. DOI: 10.1109/SP.2019.00052.

[28] Dongdong She et al. "MTFuzz: fuzzing with a multi-task neural network". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 737–749. ISBN: 9781450370431. DOI: 10.1145/3368089.3409723.

[29] Mingyuan Wu et al. "Evaluating and Improving Neural Program-Smoothing-based Fuzzing". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 847–858. DOI: 10.1145/3510003.3510089.

[30] Yuyang Rong et al. "Valkyrie: Improving Fuzzing Performance Through Deterministic Techniques". In: *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*. IEEE, 2022, pp. 628–639. DOI: 10.1109/QRS57517.2022.00069.

[31] Shuitao Gan et al. "CollAFL: Path Sensitive Fuzzing". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 679–696. DOI: 10.1109/SP.2018.00040.

[32] Jiang Zhang et al. "SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 479–494. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/zhang.

[33] Mingzhe Wang et al. "RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July

2021, pp. 147–159. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/wang-mingzhe.

[34] Chin-Chia Hsu et al. "Instrim: Lightweight instrumentation for coverage-guided fuzzing". In: *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*. Vol. 40. 2018.

[35] Jinghan Wang et al. "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 1–15. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/wang.

[36] Cornelius Aschermann et al. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[37] Alexandre Rebert et al. "Optimizing Seed Selection for Fuzzing". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 861–875. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert.

[38] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506. DOI: 10.1109/TSE.2017.2785841.

[39] Chenyang Lyu et al. "MOPT: Optimized Mutation Scheduling for Fuzzers". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[40] Jinghan Wang, Chengyu Song, and Heng Yin. "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing". In: *Network and Distributed System Security Sym-*

posium (NDSS). 2021. DOI: 10.14722/ndss.2021.24486. URL: https://par.nsf.gov/biblio/10313742.

[41] Han Zheng et al. "FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1343–1360. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/zheng.

[42] Dongdong She, Abhishek Shah, and Suman Jana. "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 2194–2211. DOI: 10.1109/SP46214.2022.9833761.

[43] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[44] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[45] Mingyuan Wu et al. "One Fuzzing Strategy to Rule Them All". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1634–1645. DOI: 10.1145/3510003.3510174.

[46] Dominic Steinhöfel and Andreas Zeller. "Language-Based Software Testing". In: *Commun. ACM* 67.4 (Mar. 2024), pp. 80–84. ISSN: 0001-0782. DOI: 10.1145/3631520. URL: https://doi.org/10.1145/3631520.

[47] Dokyung Song et al. "SoK: Sanitizing for Security". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[48] Sebastian Österlund et al. "ParmeSan: Sanitizer-guided Greybox Fuzzing". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2289–2306. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund.

[49] Will Dietz et al. “Understanding Integer Overflow in C/C++”. In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (Dec. 2015). ISSN: 1049-331X. DOI: 10.1145/2743019.

[50] Will Dietz et al. “Understanding integer overflow in C/C++”. In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 760–770. DOI: 10.1109/ICSE.2012.6227142.

[51] *Software Assurance Reference Dataset*. Accessed: 2024-03-24. Nov. 3, 2017. URL: https://samate.nist.gov/SARD/testsuite.php.

[52] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. “Magma: A Ground-Truth Fuzzing Benchmark”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (Nov. 2020). DOI: 10.1145/3428334.

[53] Andrea Fioraldi et al. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[54] Kostya Serebryany. “OSS-Fuzz - Google’s continuous fuzzing service for open source software”. In: Vancouver, BC: USENIX Association, Aug. 2017.

[55] Cornelius Aschermann et al. “NAUTILUS: Fishing for Deep Bugs with Grammars.” In: *NDSS*. 2019.

[56] Soyeon Park et al. “Fuzzing JavaScript Engines with Aspect-preserving Mutation”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1629–1642. DOI: 10.1109/SP40000.2020.00067.

[57] Bahruz Jabiyev et al. “FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1061–1075. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/jabiyev.

[58] Samuel Groß et al. “FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities.” In: *NDSS*. 2023.

[59] Chris Lattner and Vikram Adve. “LLVM: a compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[60] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[61] Alan Mathison Turing et al. “On computable numbers, with an application to the Entscheidungsproblem”. In: *J. of Math* 58.345-363 (1936), p. 5.

[62] Junjie Wang et al. “Software Testing with Large Language Models: Survey, Landscape, and Vision”. In: *IEEE Transactions on Software Engineering* (2024), pp. 1–27. DOI: 10.1109/TSE.2024.3368208.

[63] Yangruibo Ding et al. *Vulnerability Detection with Code Language Models: How Far Are We?* Accessed: 2024-04-03. 2024. arXiv: 2403.18624 [cs.SE].

[64] Xin-Cheng Wen et al. *SCALE: Constructing Structured Natural Language Comment Trees for Software Vulnerability Detection*. 2024. arXiv: 2403.19096 [cs.SE].

[65] Srinivasan Iyer et al. “Summarizing Source Code using a Neural Attention Model”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. DOI: 10.18653/v1/P16-1195.

[66] Jeffrey Svajlenko et al. “Towards a Big Data Curated Benchmark of Inter-project Code Clones”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 476–480. DOI: 10.1109/ICSME.2014.77.

[67] Lili Mou et al. "Convolutional neural networks over tree structures for programming language processing". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 1287–1293.

[68] Ruchir Puri et al. *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*. Accessed: 2024-04-03. 2021. arXiv: 2105.12655 [cs.SE].

[69] Bob Martin et al. "2011 CWE/SANS top 25 most dangerous software errors". In: *Common Weakness Enumer* 7515 (2011). URL: https://www.sans.org/top25-software-errors/.

[70] *CWE - Common Weakness Enumeration*. Accessed: 2024-03-24. Apr. 3, 2018. URL: https://cwe.mitre.org/.

[71] *BatchOverflow Exploit Creates Trillions of Ethereum Tokens, Major Exchanges Halt ERC20 Deposits — CryptoSlate*. Accessed: 2024-03-24. Apr. 25, 2018. URL: https://cryptoslate.com/batchoverflow-exploit-creates-trillions-of-ethereum-tokens/.

[72] *BeautyChain (BEC) Withdrawal and Trading Suspended*. Accessed: 2024-03-24. Apr. 24, 2018. URL: https://support.okex.com/hc/en-us/articles/360002944212-BeautyChain-BEC-Withdrawal-and-Trading-Suspended-Update-.

[73] *LLVM UndefinedBehaviorSanitizer*. Accessed: 2024-03-24. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[74] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing." In: *NDSS*. Vol. 17. 2017, pp. 1–14.

[75] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: Fuzzing by Program Transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[76] Tielei Wang et al. "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 497–512. DOI: 10.1109/SP.2010.37.

[77] Marcel Böhme et al. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. ISBN: 9781450349468. DOI: 10.1145/3133956.3134020.

[78] Hongxu Chen et al. "Hawkeye: Towards a Desired Directed Grey-box Fuzzer". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2095–2108. ISBN: 9781450356930. DOI: 10.1145/3243734.3243849.

[79] *The GNU Multiple Precision Arithmetic Library*. Accessed: 2024-03-24. 2016. URL: https://gmplib.org/.

[80] *LLVM DataFlowSanitizer*. Accessed: 2024-03-24. URL: https://clang.llvm.org/docs/DataFlowSanitizer.html.

[81] Hao Sun et al. "IntEQ: Recognizing Benign Integer Overflows via Equivalence Checking across Multiple Precisions". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 1051–1062. DOI: 10.1145/2884781.2884820.

[82] Marios Pomonis et al. "IntFlow: improving the accuracy of arithmetic error detection using information flow tracking". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM. 2014, pp. 416–425.

[83] Yannick Moy, Nikolaj Bjørner, and David Sielaff. "Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis". In: *Microsoft Research* 11 (2009).

[84] Tielei Wang et al. "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution." In: *NDSS*. Citeseer. 2009.

[85] Peng Chen, Jianzhong Liu, and Hao Chen. "Matryoshka: Fuzzing Deeply Nested Branches". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 499–513. ISBN: 9781450367479. DOI: 10.1145/3319535.3363225.

[86] Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 157–157. DOI: 10.1109/SecDev.2016.043.

[87] Nick Stephens et al. "Driller: augmenting fuzzing through selective symbolic execution". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2016.

[88] Insu Yun et al. "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[89] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[90] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: Fast detector of uninitialized memory use in C++". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186.

[91] Wookhyun Han et al. "Enhancing memory error detection for large-scale applications and fuzz testing". In: *Network and Distributed Systems Security (NDSS) Symposium 2018*. 2018.

[92] *LLVM ThreadSanitizer*. Accessed: 2024-03-24. URL: https://clang.llvm.org/docs/ThreadSanitizer.html.

[93] Theofilos Petsios et al. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2155–2168. ISBN: 9781450349468. DOI: 10.1145/3133956.3134073.

[94] Theofilos Petsios et al. “NEZHA: Efficient Domain-Independent Differential Testing”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 615–632. DOI: 10.1109/SP.2017.27.

[95] Augustus Odena et al. “Tensorfuzz: Debugging neural networks with coverage-guided fuzzing”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4901–4911.

[96] Istvan Haller et al. “Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 49–64. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller.

[97] Vivek Jain et al. “TIFF: Using Input Type Inference To Improve Fuzzing”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC ’18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 505–517. ISBN: 9781450365697. DOI: 10.1145/3274694.3274746.

[98] Baozheng Liu et al. “FANS: Fuzzing Android Native System Services via Automated Interface Analysis”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 307–323. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/liu.

[99] Andreas Zeller. *When Results Are All That Matters: The Case of the Angora Fuzzer*. Accessed: 2024-03-24. Oct. 2019. URL: https://andreas-zeller.info/2019/10/10/when-results-are-all-that-matters-case.html.

[100] George Klees et al. “Evaluating Fuzz Testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. ISBN: 9781450356930. DOI: 10.1145/3243734.3243804.

[101] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. “Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers”. In: *Proceedings of the ACM*

SIGPLAN 1994 Conference on Programming Language Design and Implementation. PLDI '94. Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 242–256. ISBN: 089791662X. DOI: 10.1145/178243.178264.

[102] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 32–41. ISBN: 0897917693. DOI: 10.1145/237721.237727.

[103] *gllvm: Whole Program LLVM in Go*. Accessed: 2024-03-24. URL: https://github.com/SRI-CSL/gllvm.

[104] Jonathan Metzman et al. "FuzzBench: an open fuzzer benchmarking platform and service". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1393–1403. ISBN: 9781450385626. DOI: 10.1145/3468264.3473932.

[105] Yuwei Li et al. "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2777–2794. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei.

[106] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[107] Yuyang Rong, Peng Chen, and Hao Chen. "Integrity: Finding Integer Errors by Targeted Fuzzing". In: *Security and Privacy in Communication Networks - 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part I*. Ed. by Noseong Park et al. Vol. 335. Lecture Notes of the Institute for Computer

Sciences, Social Informatics and Telecommunications Engineering. Springer, 2020, pp. 360–380. DOI: 10.1007/978-3-030-63086-7_20.

[108] *afl-cov*. Accessed: 2024-03-24. URL: https://github.com/mrash/afl-cov.

[109] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[110] Yaohui Chen et al. "SAVIOR: Towards Bug-Driven Hybrid Testing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1580–1596. DOI: 10.1109/SP40000.2020.00002.

[111] Zi Wang, Ben Liblit, and Thomas W. Reps. "TOFU: Target-Orienter FUzzer". In: *CoRR* abs/2004.14375 (2020). arXiv: 2004.14375. URL: https://arxiv.org/abs/2004.14375.

[112] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 579–594. DOI: 10.1109/SP.2017.23.

[113] Xiao Liu et al. "DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing". In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'19/IAAI'19/EAAI'19. Honolulu, Hawaii, USA: AAAI Press, 2019. ISBN: 978-1-57735-809-1. DOI: 10.1609/aaai.v33i01.33011044.

[114] Yinlin Deng et al. "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 423–435. ISBN: 9798400702211. DOI: 10.1145/3597926.3598067.

[115] Chunqiu Steven Xia et al. *Fuzz4All: Universal Fuzzing with Large Language Models*. Accessed: 2024-04-03. 2024. arXiv: 2308.04748 [cs.SE].

[116] Jianyu Zhao et al. "Understanding Programs by Exploiting (Fuzzing) Test Cases". In: *ACL* (2023). URL: https://aclanthology.org/2023.findings-acl.678.pdf.

[117] Jie Hu, Qian Zhang, and Heng Yin. *Augmenting Greybox Fuzzing with Generative AI*. Accessed: 2024-04-03. 2023. arXiv: 2306.06782 [cs.CR].

[118] Dawei Wang et al. "CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1919–1936. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/wang-dawei.

[119] Nuno P. Lopes et al. "Alive2: Bounded Translation Validation for LLVM". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 65–79. DOI: 10.1145/3453483.3454030.

[120] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1051–1065. DOI: 10.1145/3548606.3560602.

[121] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Random Testing for C and C++ Compilers with YARPGen". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428264.

[122] 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*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 216–226. DOI: 10.1145/2594291.2594334.

[123] 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. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 347–361. DOI: 10.1145/3062341.3062379.

[124] Jacques-Henri Jourdan et al. "A Formally-Verified C Static Analyzer". In: *POPL 2015: 42nd symposium Principles of Programming Languages*. ACM Press, 2015, pp. 247–259. URL: http://xavierleroy.org/publi/verasco-popl2015.pdf.

[125] Timothy Bourke et al. "A formally verified compiler for Lustre". In: *PLDI 2017: Programming Language Design and Implementation*. ACM Press, 2017, pp. 586–601. URL: http://xavierleroy.org/publi/velus-pldi17.pdf.

[126] *Fuzzing LLVM libraries and tools*. Accessed: 2024-04-03.

[127] Justin Bogner. *Adventures in Fuzzing Instruction Selection*. Accessed: 2024-04-03. Mar. 2017.

[128] Seo Sanghyeon. *Rust triggers LLVM ARM backend bug*. Accessed: 2024-04-03. 2013.

[129] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[130] Reese T. Prosser. "Applications of Boolean matrices to the analysis of flow diagrams". In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). Boston, Massachusetts: Association for Computing Machinery, 1959, pp. 133–138. ISBN: 9781450378680. DOI: 10.1145/1460299.1460314.

[131] *The LLVM Target-Independent Code Generator*. Accessed: 2024-04-03.

[132] *GlobalIsel*. Accessed: 2024-04-03.

[133] *TableGen Overview*. Accessed: 2024-04-03.

[134] Xiaogang Zhu et al. “Fuzzing: A Survey for Roadmap”. In: *ACM Comput. Surv.* 54.11s (Sept. 2022). DOI: 10.1145/3512345.

[135] Junjie Wang et al. “Superion: Grammar-Aware Greybox Fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 724–735. DOI: 10.1109/ICSE.2019.00081.

[136] Samuel Groß. “Fuzzil: Coverage guided fuzzing for javascript engines”. In: *Department of Informatics, Karlsruhe Institute of Technology* (2018).

[137] Anonymous. *IRFuzzer artifacts*. Accessed: 2024-04-03. Zenodo, Mar. 2024. DOI: 10.5281/zenodo.8388300.

[138] Karine Even-Mendoza et al. *Artifact of GrayC: Greybox Fuzzing of Compilers and Analysers for C*. Version GrayC-ISSTA-2023-V1.0. Accessed: 2024-04-03. July 2023. DOI: 10.5281/zenodo.7978251.

[139] Junjie Chen et al. “A Survey of Compiler Testing”. In: *ACM Comput. Surv.* 53.1 (Feb. 2020). DOI: 10.1145/3363562.

[140] Haoyang Ma. *A Survey of Modern Compiler Fuzzing*. Accessed: 2024-04-03. 2023. arXiv: 2306.06884 [cs.SE].

[141] Michaël Marcozzi et al. “Compiler Fuzzing: How Much Does It Matter?” In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360581.

[142] Paul Purdom. “A sentence generator for testing parsers”. In: *BIT Numerical Mathematics* 12.3 (1972), pp. 366–375.

[143] Andreas Zeller et al. *The Fuzzing Book*. Retrieved 2024-01-18 17:28:37+01:00. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/.

[144] William Mansky and Elsa Gunter. “A framework for formal verification of compiler optimizations”. In: *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 1*. Springer. 2010, pp. 371–386.

[145] Nuno P. Lopes et al. “Practical Verification of Peephole Optimizations with Alive”. In: *Commun. ACM* 61.2 (Jan. 2018), pp. 84–91. DOI: 10.1145/3166064.

[146] Kyle Dewey, Jared Roesch, and Ben Hardekopf. “Fuzzing the Rust Typechecker Using CLP (T)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 482–493. DOI: 10.1109/ASE.2015.65.

[147] Yuting Chen et al. “Coverage-Directed Differential Testing of JVM Implementations”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 85–99. DOI: 10.1145/2908080.2908095.

[148] Emin Gün Sirer and Brian N. Bershad. “Using Production Grammars in Software Testing”. In: *Proceedings of the 2nd Conference on Domain-Specific Languages*. DSL ’99. Austin, Texas, USA: Association for Computing Machinery, 2000, pp. 1–13. DOI: 10.1145/331960.331965.

[149] Mingyuan Wu et al. “JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers”. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE ’23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 56–68. DOI: 10.1109/ICSE48619.2023.00017.

[150] Yongheng Chen et al. “One Engine to Fuzz ’em All: Generic Language Processor Testing with Semantic Validation”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 642–658. DOI: 10.1109/SP40001.2021.00071.

[151] 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*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 386–399. DOI: 10.1145/2814270.2814319.

[152] Vu Le, Chengnian Sun, and Zhendong Su. "Finding Deep Compiler Bugs via Guided Stochastic Program Mutation". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 386–399. DOI: 10.1145/2858965.2814319.

[153] Christopher Lidbury et al. "Many-Core Compiler Fuzzing". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 65–76. DOI: 10.1145/2737924.2737986.

[154] Maulik A. Dave. "Compiler Verification: A Bibliography". In: *SIGSOFT Softw. Eng. Notes* 28.6 (Nov. 2003), p. 2. DOI: 10.1145/966221.966235.

[155] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[156] Vytautas Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360573.

[157] Haoyang Ma et al. "Fuzzing Deep Learning Compilers with HirGen". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 248–260. DOI: 10.1145/3597926.3598053.

[158] Jiawei Liu et al. "Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527317.

[159] Yuting Chen, Ting Su, and Zhendong Su. "Deep Differential Testing of JVM Implementations". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1257–1268. DOI: 10.1109/ICSE.2019.00127.

[160] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[161] Yujia Li et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (2022), pp. 1092–1097. DOI: 10.1126/science.abq1158. eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158. URL: https://www.science.org/doi/abs/10.1126/science.abq1158.

[162] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Addison-Wesley Professional, 2007. ISBN: 0321446119.

[163] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *J. Mach. Learn. Res.* 21.1 (Jan. 2020). ISSN: 1532-4435.

[164] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. Accessed: 2024-04-03. 2019. arXiv: 1907.11692 [cs.CL].

[165] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[166] Aditya Kanade et al. "Learning and evaluating contextual embedding of source code". In: *Proceedings of the 37th International Conference on Machine Learning*. ICML'20. JMLR.org, 2020.

[167] Yue Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. Accessed: 2024-04-03. 2021. arXiv: 2109.00859 [cs.CL].

[168] Shuai Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. Accessed: 2024-04-03. 2021. arXiv: 2102.04664 [cs.SE].

[169] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. "StructCoder: Structure-Aware Transformer for Code Generation". In: *ACM Trans. Knowl. Discov. Data* 18.3 (Jan. 2024). ISSN: 1556-4681. DOI: 10.1145/3636430.

[170] Daya Guo et al. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. Accessed: 2024-04-03. 2022. arXiv: 2203.03850 [cs.CL].

[171] Daya Guo et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow". In: *CoRR* abs/2009.08366 (2020). arXiv: 2009.08366. URL: https://arxiv.org/abs/2009.08366.

[172] Dinglan Peng et al. "How could Neural Networks understand Programs?" In: *International Conference on Machine Learning*. PMLR. 2021, pp. 8476–8486.

[173] Jubi Taneja, Zhengyang Liu, and John Regehr. "Testing Static Analyses for Precision and Soundness". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 81–93. ISBN: 9781450370479. DOI: 10.1145/3368826.3377927.

[174] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. "POSTER: AFL-Based Fuzzing for Java with Kelinci". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2511–2513. ISBN: 9781450349468. DOI: 10.1145/3133956.3138820.

[175] Fabio Petroni et al. "Language Models as Knowledge Bases?" In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 2463–2473. DOI: 10.18653/v1/D19-1250.

[176] Adam Paszke et al. "PyTorch: an imperative style, high-performance deep learning library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[177] Wasi Ahmad et al. "Unified Pre-training for Program Understanding and Generation". In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Ed. by Kristina Toutanova

et al. Online: Association for Computational Linguistics, June 2021, pp. 2655–2668. DOI: 10.18653/v1/2021.naacl-main.211.

[178] Xin Wang et al. *SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation*. Accessed: 2024-04-03. 2021. arXiv: 2108.04556 [cs.CL].

[179] Shangqing Liu et al. "ContraBERT: Enhancing Code Pre-Trained Models via Contrastive Learning". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 2476–2487. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00207.

[180] Chris Cummins et al. *ProGraML: Graph-based Deep Learning for Program Optimization and Analysis*. Accessed: 2024-04-03. 2020. arXiv: 2003.10536 [cs.LG].

[181] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. "A metric learning reality check". In: *European Conference on Computer Vision*. Springer. 2020, pp. 681–699.

[182] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Accessed: 2024-04-03. 2017. arXiv: 1412.6980 [cs.LG].

[183] Kyriakos Ispoglou et al. "FuzzGen: Automatic Fuzzer Generation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[184] Hamel Husain et al. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. Accessed: 2024-04-03. 2020. arXiv: 1909.09436 [cs.LG].

[185] Chenxiao Liu et al. "Code Execution with Pre-trained Language Models". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 4984–4999. DOI: 10.18653/v1/2023.findings-acl.308.

[186] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. "Software clone detection: A systematic review". In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.01.008. URL: https://www.sciencedirect.com/science/article/pii/S0950584913000323.

[187] Aman Madaan et al. *Language Models of Code are Few-Shot Commonsense Learners*. Accessed: 2024-04-03. 2022. arXiv: 2210.07128 [cs.CL].

[188] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[189] Brian A Wichmann et al. "Industrial perspective on static analysis". In: *Software Engineering Journal* 10.2 (1995), p. 69.

[190] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=BJOFETxR-.

[191] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[192] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

[193] Vincent J. Hellendoorn et al. "Global Relational Models of Source Code". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=B1lnbRNtwr.

[194] Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. "Just enough semantics: An information theoretic approach for IR-based software bug localization". In: *Information and Software Technology* 93 (2018), pp. 45–57. ISSN: 0950-5849. DOI: https://doi.org/10.1016/

j.infsof.2017.08.012. URL: https://www.sciencedirect.com/science/article/pii/S0950584916302269.

[195] Max Schäfer et al. "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation". In: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 85–105. DOI: 10.1109/TSE.2023.3334955.

[196] Sungmin Kang, Juyeon Yoon, and Shin Yoo. "Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 2312–2323. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00194.

[197] Ke Wang, Rishabh Singh, and Zhendong Su. *Dynamic Neural Program Embedding for Program Repair*. Accessed: 2024-04-03. 2018. arXiv: 1711.07163 [cs.AI].

[198] Ke Wang and Zhendong Su. "Blended, precise semantic program embeddings". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 121–134. ISBN: 9781450376136. DOI: 10.1145/3385412.3385999.

[199] Jordan Henkel et al. "Code vectors: understanding programs through embedded abstracted symbolic traces". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 163–174. ISBN: 9781450355735. DOI: 10.1145/3236024.3236085.

[200] Yangruibo Ding et al. "TRACED: Execution-aware Pre-training for Source Code". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3608140.

[201] Eui Chul Shin, Illia Polosukhin, and Dawn Song. "Improving Neural Program Synthesis with Inferred Execution Traces". In: *Advances in Neural Information Processing Systems*. Ed. by

S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/7776e88b0c189539098176589250bcba-Paper.pdf.

[202] Xinyun Chen, Dawn Song, and Yuandong Tian. "Latent Execution for Neural Program Synthesis Beyond Domain-Specific Languages". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 22196–22208. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/ba3c95c2962d3aab2f6e667932daa3c5-Paper.pdf.

[203] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. Accessed: 2024-04-03. 2016. arXiv: 1609.08144 [cs.CL].

[204] Yonglong Tian, Dilip Krishnan, and Phillip Isola. *Contrastive Representation Distillation*. Accessed: 2024-04-03. 2022. arXiv: 1910.10699 [cs.LG].

[205] Kaiming He et al. "Momentum Contrast for Unsupervised Visual Representation Learning". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 9726–9735. DOI: 10.1109/CVPR42600.2020.00975.

[206] James M Lucas and Michael S Saccucci. "Exponentially weighted moving average control schemes: properties and enhancements". In: *Technometrics* 32.1 (1990), pp. 1–12.

[207] Alexis Conneau and Guillaume Lample. "Cross-lingual language model pretraining". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[208] Yaqin Zhou et al. "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[209] Junjie Huang et al. "CoSQA: 20,000+ Web Queries for Code Search and Question Answering". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long*

Papers). Ed. by Chengqing Zong et al. Online: Association for Computational Linguistics, Aug. 2021, pp. 5690–5700. DOI: 10.18653/v1/2021.acl-long.442.

[210] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[211] Tianyu Gao, Xingcheng Yao, and Danqi Chen. "SimCSE: Simple Contrastive Learning of Sentence Embeddings". In: Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 6894–6910. DOI: 10.18653/v1/2021.emnlp-main.552.

[212] Yangruibo Ding et al. "Towards Learning (Dis)-Similarity of Source Code from Program Contrasts". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 6300–6312. DOI: 10.18653/v1/2022.acl-long.436.

[213] Xiaonan Li et al. "CodeRetriever: A Large Scale Contrastive Pre-Training Method for Code Search". In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 2898–2910. DOI: 10.18653/v1/2022.emnlp-main.187.

[214] Yue Wang et al. "CodeT5+: Open Code Large Language Models for Code Understanding and Generation". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1069–1088. DOI: 10.18653/v1/2023.emnlp-main.68.

[215] Xiaonan Li et al. "Soft-Labeled Contrastive Pre-Training for Function-Level Code Representation". In: *Findings of the Association for Computational Linguistics: EMNLP 2022*. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates:

Association for Computational Linguistics, Dec. 2022, pp. 118–129. DOI: 10.18653/v1/2022.findings-emnlp.9.

[216] Chunqiu Steven Xia et al. "Fuzz4All: Universal Fuzzing with Large Language Models". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. ¡conf-loc¿, ¡city¿Lisbon¡/city¿, ¡country¿Portugal¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639121. URL: https://doi.org/10.1145/3597503.3639121.

[217] Ruijie Meng et al. "Large language model guided protocol fuzzing". In: *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*. 2024.

[218] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. *CovRL: Fuzzing JavaScript Engines with Coverage-Guided Reinforcement Learning for LLM-based Mutation*. Accessed: 2024-05-08. 2024. arXiv: 2402.12222 [cs.CR].

[219] Chenyuan Yang et al. *White-box Compiler Fuzzing Empowered by Large Language Models*. Accessed: 2024-05-08. 2023. arXiv: 2310.15991 [cs.SE].

[220] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. Accessed: 2024-05-08. 2023. arXiv: 2307.09288 [cs.CL].