DOCTORAL THESIS

Advancing Software Reliability from Code to Compilation

Shaohua Li

DISS. ETH NO. 30279

DISS. ETH NO. 30279

ADVANCING SOFTWARE RELIABILITY FROM CODE TO COMPILATION

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zürich)

presented by

SHAOHUA LI

M. eng., University of Science and Technology of China

born on 8 December 1994

accepted on the recommendation of
Prof. Dr. Zhendong Su, examiner
Prof. Dr. Mathias Payer, co-examiner
Prof. Dr. Andreas Zeller, co-examiner

2024

ABSTRACT

Software takes charge of every critical aspect of our modern society, including communication, finance, transportation, and many more. It is thus crucial to ensure the reliability of software systems. Yet, guaranteeing that non-trivial software systems are free of defects is extremely difficult, if not impossible. Consequently, modern software systems are full of bugs, such as security vulnerabilities, semantic bugs, performance issues, *etc.*

The motivating question of this thesis is: *where can software go wrong?* Software development is an intricate process with many different procedures in the pipeline. Beyond the source code written by developers, there are many other tools involved, such as code analysis tools used for identifying defects and compilers used for translating source code into machine code. Unfortunately, they can all go wrong. In this thesis, we study the reliability problem from three different levels: *code*, *code analysis*, and *code compilation*. At a high level, we design new methodologies to identify and detect bugs at all of these levels.

For the reliability of code, we focus on eliminating undefined behavior, a major source of reliability bugs such as buffer-overflow and use-after-free, in modern C/C++ software. We develop a general detection approach to identify undefined behaviors practically and effectively. To improve detection efficiency, we further present two novel concepts to accelerate the existing detection frameworks. **For the reliability of code analysis**, we aim to validate existing bug detection tools for undefined behaviors. We propose and design the first program generator that can automatically produce a large number of programs with various undefined behaviors. We then use this generator to validate sanitizers, one of the most popular toolsets for undefined behavior detection. **For the reliability of code compilation**, we concentrate on solidifying the modern compiler implementations. We introduce a novel data-driven program generation technique that can generate expressive and well-formed programs based on real-world code snippets.

At the conceptual level, this thesis highlights the prevalence of reliability problems in the software development pipeline, from code to compilation. At the technical level, this thesis presents five new tools for detecting software defects in source code, code analysis tools, and compilers.

ZUSAMMENFASSUNG

Software kontrolliert jeden Aspekt unserer modernen Gesellschaft, wie Kommunikation, Finanzen, Verkehr und vieles mehr. Es ist daher entscheidend, die Zuverlässigkeit von Softwaresystemen zu gewährleisten. Es ist jedoch schwierig, oft unmöglich, die Zuverlässigkeit von nicht-trivialer Software aufgrund ihrer enormen Komplexität zu garantieren. Als Ergebnis sind moderne Softwaresysteme voller Bugs, wie z.B. Sicherheitslücken, semantische Bugs oder Leistungsprobleme. Die Leitfrage dieser Dissertation lautet: Was kann bei Software schiefgehen? Softwareentwicklung ist ein komplexer Prozess mit vielen verschiedenen Abläufen. Neben dem von Entwicklern geschriebenen Quellcode sind viele andere Hilfsmittel involviert, wie Codeanalysetools und Compiler, die zur Identifizierung von Defekten im Quellcode respektive zur Übersetzung von Quellcode in Maschinencode verwendet werden. Alle diese Hilfsmittel können Bugs enthalten. In dieser Dissertation untersuchen wir die Zuverlässigkeitsprobleme auf drei verschiedenen Ebenen: Code, Codeanalyse und Codekompilierung. Auf konzeptioneller Ebene entwerfen wir neue Methoden, um Fehler auf all diesen Ebenen zu identifizieren und zu erkennen. Um die Zuverlässigkeit des Codes zu verbessern, konzentrieren wir uns auf Undefined Behavior. Undefined Behavior ist eine der Hauptquellen für Zuverlässigkeitsfehler wie Buffer Overflows und use-after-free Bug in modernen C/C++ Software Systemen. Wir entwickeln eine allgemeine Erkennungsmethode, um Undefined Behavior effektiv zu identifizieren. Um die Erkennungseffizienz zu verbessern, präsentieren wir zusätzlich zwei neue Strategien zur Beschleunigung der bestehenden Erkennungsmethode. Für die Zuverlässigkeit der Codeanalyse zielen wir darauf ab, die Robustheit vorhandener Bugerkennungstools für Undefined Behavior zu validieren. Wir entwerfen und entwickeln den ersten Programmgenerator, der automatisch eine grosse Anzahl von Programmen mit verschiedenen Arten von Undefined Behavior produzieren kann. Anschließend verwenden wir diesen Generator, um Sanitizer, eines der gängigsten Toolsets zur Erkennung von Undefined Behavior, zu validieren. Für die Zuverlässigkeit der Codekompilierung konzentrieren wir uns darauf, die moderne Compilerinfrastruktur zu verbessern. Wir präsentieren eine neuartige daten-basierte Programmgenerierungstechnik, die vielfältige, komplexe und wohlgeformte Programme erzeugen kann, mit denen wir moderne Compiler testen. Auf konzeptioneller Ebene hebt diese

Dissertation die Verbreitung von Zuverlässigkeitsproblemen im Softwareentwicklungsprozess hervor, insbesondere von Code bis zur Kompilierung. Auf technischer Ebene präsentiert diese Dissertation fünf neue, öffentlich verfügbare Tools zur Erkennung von Softwaredefekten in Quellcode, Codeanalysewerkzeugen und Compilern.

ACKNOWLEDGEMENTS

Five years ago, I landed in Zurich, thousands of miles away from my hometown, filled with excitement and dreams. Now, I stand at the completion of my doctorate journey at ETH Zurich. It has been both bitter and sweet, exhausting and exciting, tough and rewarding. Looking back, I feel a deep sense of satisfaction and thankfulness. This thesis is not just my work; it is a collection of countless pieces of advice, support, and encouragement that I got directly and indirectly from many people along the way. I would like to use this page to thank them all.

First and foremost, my deepest appreciation goes to my advisor, Zhendong Su, whose advice, encouragement, and unwavering support have been the cornerstone of this journey. His insight, vision, attitude, and approach to research have shaped not just this work but my growth as an independent researcher and thinker. Working with him has been an honor, a privilege, and a joy.

I would also like to extend my sincere thanks to Andreas Zeller and Mathias Payer for dedicating your time to be part of my defense committee.

Being a part of the AST Lab has been an amazing ride. I want to express my gratitude to my colleagues, past and present, who have fueled and inspired my research with hours-long discussions and brainstorming sessions: Manuel Rigger, Zu-Ming Jiang, Chengyu Zhang, Theodoros Theodoridis, Dominik Winterer, Yann Girsberger, Michel Weber, Hao Sun, Heqing Huang, and Thodoris Sotiropoulos. Also, a warm thank you to all my friends and colleagues in the lab with whom I have shared both time and memories.

A special word of thanks goes to the administrative team in our lab: Ariane Nake, Tracy Ewen, and Christian Rossi. Ari, your assistance during my early days in Zurich was invaluable. Tracy, thank you for arranging various teaching stuff and supporting my visa applications. Chris, thank you for handling all my reimbursement requests and keeping us stocked with coffee and tea.

Finally, I want to thank my family for their constant support. Most importantly, I want to express my heartfelt thank you to my partner, Yujia Liu, for her enduring support and endless love over the past eleven years. Without you, I would not have reached where I am now. Thank you for being by my side for every up and down.

PUBLICATIONS

This thesis is based on the following publications:

- **Shaohua Li**, Zhendong Su.
  Finding Unstable Code via Compiler-Driven Differential Testing.
  ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023.

- **Shaohua Li**, Zhendong Su.
  Accelerating Fuzzing through Prefix-guided Execution.
  ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2023.

- **Shaohua Li**, Zhendong Su.
  UBFuzz: Finding Bugs in Sanitizer Implementations.
  ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024.

- **Shaohua Li**, Theodoros Theodoridis, Zhendong Su.
  Boosting Compiler Testing by Injecting Real-world Code.
  ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2024.

- Ziqiao Kong*, **Shaohua Li***, Heqing Huang, Zhendong Su.
  SAND: Decoupling Sanitization from Fuzzing for Low Overhead.
  arXiv preprint arXiv:2402.16497, 2024. (*Equal contributions.)

The following publication was part of my doctorate research, but is not covered in this thesis:

- Jue Wang, Yanyan Jiang, Ting Su, **Shaohua Li**, Chang Xu, Jian Lu, Zhendong Su.
  Detecting Non-crashing Functional Bugs in Android Apps via Deep-State Differential Analysis.
  ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022.

CONTENTS

1 INTRODUCTION

Software has taken every critical aspect of our modern society, such as communication, transportation, entertainment, education, finance, and many more. Due to the indispensability of software, we are increasingly valuing its reliability. Software reliability problems have led to considerable problems in our modern society. For instance, a public report [24] from The Consortium for IT Software Quality (CISQ) group in 2022 comments that "we estimate that the cost of poor software quality in the US has grown to at least $2.41 trillion." Software reliability problems often result from various software defects, such as functional bugs, security vulnerabilities, and runtime faults, to name a few. These defects can lead to errors in many aspects of software systems, including correctness, security, performance, *etc.* To tackle this ever-growing problem, countless efforts are made in both academia and industry to effectively and efficiently detect software defects. Unfortunately, it is hard, often impossible, to guarantee the complete correctness of modern software systems due to their high complexity.

MOTIVATION The motivation question of this thesis is: *where can software go wrong?* Suppose we are now developing a new piece of software. We start by writing code for it. Then, in order to improve or ensure the correctness of our code, we may utilize powerful code analysis tools, such as bug detectors, to analyze our code to identify potential defects. After we fix all the reported issues, we consider the code to be ready for deployment. We then use a compiler to translate our code written in high-level programming languages into low-level machine code so that it can be deployed and run on different platforms or hardware. Suppose there is a defect in our software, if we look at the whole software development pipeline, we may wonder *where can our software go wrong?* Unfortunately, this thesis will show that all of the above processes, from code to compilation, can be wrong.

CONTRIBUTIONS In this thesis, we study the software reliability problem from three different levels: *code*, *code analysis*, and *code compilation*. At a high level, we (1) introduce novel detection methodologies for effectively and efficiently identifying software defects and (2) reveal previously unknown

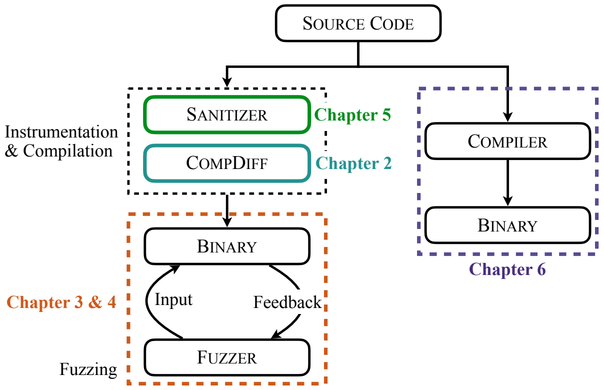


Figure 1.1: Overview of chapters.

or overlooked reliability problems. Specifically, the contributions of this thesis can be summarized as follows:

- *Reliability of code* (Chapter 2, Chapter 3, and Chapter 4): At the code level, we introduce CompDiff, a new and novel compiler-driven differential testing oracle for effectively detecting a wide range of undefined behaviors in software. To accelerate the efficiency of CompDiff as well as other oracles, we further propose PGE and Sand, two acceleration frameworks to reduce the execution or sanitization redundancy during fuzzing.

- *Reliability of code analysis* (Chapter 5): At the code analysis level, we reveal the false negative problems of sanitizers, a set of the most popular dynamic bug detectors that are broadly applied to detect security vulnerabilities. We present the first validation framework, UBfuzz, to validate sanitizers' reliability on various automatically generated programs that contain security vulnerabilities.

- *Reliability of code compilation* (Chapter 6): At the code compilation level, we propose a novel data-driven program generation approach, Creal, for extensively testing compilers. Creal breaks the constraints of existing random program generators and boosts their expressiveness by injecting real-world code snippets into random seed programs.

Overall, this thesis presents a set of new methodologies for detecting various software defects to improve the reliability of our modern software systems. Figure 1.1 shows the connections between chapters or contributions. In the following, we first introduce the background knowledge and then provide a brief overview of each contribution.

1.1 BACKGROUND

UNDEFINED BEHAVIORS Some programming languages, such as C/C++, designate a set of code constructs as having undefined behavior (UB) to simplify the compiler implementation. For example, the C17 standard [14] lists 211 circumstances for which it invokes undefined behavior. According to the standard, permissible undefined behavior results in "ignoring the situation completely with unpredictable results". Compilers can assume that undefined behavior will never occur in the program, which allows for many optimization opportunities. Undefined behaviors in the source code have notoriously led to catastrophic software reliability problems. For example, six of Mitre's CWE top 25 most dangerous software weaknesses in 2023 were undefined behaviors [114]. Well-known undefined behaviors include buffer-overflow, use-after-free, integer-overflow, *etc.* Practitioners in both academia and industry have put significant efforts into designing various static [127, 104] and dynamic tools [130, 107, 13] for the detection of undefined behaviors. Yet, it remains an unsolved problem.

SANITIZERS Sanitizers are crucial in enabling the large-scale detection of security vulnerabilities caused by UB [131, 132]. Popular sanitizers include Address Sanitizer (ASan) [130] for memory access errors, Undefined Behavior Sanitizer (UBSan) [88] for various undefined behaviors, and Memory Sanitizer (MSan) [136] for uninitialized memory uses. Technically, sanitizers are integrated into compilers. When a sanitizer is enabled, various checks are inserted into a program during compilation. If a check is violated at run-time, an error is reported. Owing to their superior capability and usability, sanitizers have assisted developers in discovering numerous critical vulnerabilities.

FUZZING Although tools like sanitizers are useful in reporting undefined behaviors, they require specific inputs that can reach the buggy code and trigger the undefined behaviors. Fuzzing [1, 96] is an effective way of generating inputs to trigger undefined behaviors. For instance, by fuzzing with sanitizers, the Google OSS-Fuzz project has reported over 20K UBs in hundreds of open-source projects [30, 49]. The bottom left of Figure 1.1

illustrates the high-level fuzzing workflow. A fuzzer first generates/mutates an input and then executes the target binary on it. If the target binary crashes on the input or sanitizers report the error, we successfully identify a bug. State-of-the-art fuzzing efforts center on coverage-guided fuzzing (CGF), *e.g.*, American Fuzzy Lop (AFL) [161]. CGF works by generating a large number of tests from mutating seed inputs. All tests are executed on a target binary, and those coverage-increasing tests will be added to the seed pool for further exploration.

COMPILERS Compilers are among the most fundamental and critical components in the software ecosystem. They should not crash and are expected to translate valid input programs faithfully. However, compiler bugs can cause crashes or, even worse, miscompilations. Decades of research and engineering efforts have been put into improving compilers' reliability [17]. Random compiler testing has proven to be effective in discovering compiler bugs. One noticeable contribution is Csmith [159], a pioneering C program generator. By generating random programs, Csmith has helped discover many bugs in C compilers. A more recent generator YARPGen [85] and the follow-up YARPGen v.2 [86], featuring a set of generation policies, target at scalar and loop optimizations and have successfully identified many bugs missed by previous generators. In addition to generator-based tools, mutation-based compiler testing techniques are also effective in finding compiler bugs. The most representative approach is *equivalence modulo inputs* (EMI) [66]. EMI techniques [67, 139] mutate a program by removing or inserting random code while maintaining the input/output behaviors of the program. This line of research has achieved a surprising result: discovering a considerable number of additional bugs in mature and well-established compilers, such as GCC and LLVM.

1.2 RELIABILITY OF CODE

Software is usually developed in one or multiple programming languages. At the source code level, there can be different types of software defects, such as semantic bugs and security bugs. In this part, we focus on *undefined behavior* (UB) and its detection.

1.2.1 *Contribution:* CompDiff

Exiting UB detectors like sanitizers, although powerful, can only cover a small range of common UBs. For example, ASan, UBSan, and MSan together only support around 20 kinds of UBs. However, as discussed above, the C17 standard [14] lists 211 kinds of UBs. How to detect a more diverse range of UBs is still an open problem.

KEY IDEA Our idea is based on the fact that undefined behaviors can lead to different binary semantics across compilations. This conforms to the standard that compilers can, in principle, do arbitrary optimizations on such erroneous code. In this thesis, we use this fact and propose a simple, straightforward, yet effective approach for finding undefined behaviors. Our approach involves three steps. First, we compile the target program with different compiler implementations to get a set of binaries. Second, run these binaries on the same set of test inputs and collect their outputs. Finally, compare outputs produced by different binaries on the same input and report discrepancies. For a program with deterministic output, *i.e.*, repeated executions on the same input always yield the same output, output discrepancy over the same input implies the presence of unstable code. We call our approach *compiler-driven differential testing* (CompDiff).

IMPLEMENTATION We implement CompDiff for C/C++ programs by integrating two compilers, *i.e.*, gcc and clang. Each compiler is configured with five different optimization levels, *i.e.*, `-O0`, `-O1`, `-Os`, `-O2`, and `-Os`. Together, there are ten compiler configurations in CompDiff. To improve CompDiff's practicality and detect undefined behaviors in real-world software, we integrate CompDiff into AFL++ [34], the most widely-used general-purpose fuzzer. Our evaluation shows that, on the Juliet benchmark tests, CompDiff uniquely identified 1,409 bugs that sanitizers failed to. On 23 popular open-source C/C++ projects, CompDiff-AFL++ discovered 78 new bugs, 66 of which were confirmed, and 52 were fixed by the developers. Of these new bugs, 36 were not detected by sanitizers. The artifact for CompDiff, including all source code and data, is permanently available [72].

1.2.2 *Contribution:* PGE

As introduced before, fuzzers can generate and execute a large number of test inputs to find bugs in the target software. However, executing all mutated test inputs incurs significant performance penalties— Many previous efforts have shown that only a tiny fraction of the generated tests increase code coverage, *e.g.*, fewer than 1 in 10,000 [105]. Significant research efforts have targeted this inefficiency, such as (1) *seed scheduling* [95, 97]: effectively select seed-to-mutate from the seed pool to prioritize seeds that are likely coverage-increasing or bug-triggering, (2) *tracing-cost reduction* [105, 106, 167]: reduce the cost of coverage tracing with sophisticated designs, and (3) *new mutations* [68, 133]: perform targeted instead of random mutations to increase the likelihood of covering new regions. However, they still need to execute a large amount of non-coverage-increasing tests fully. Thus, avoiding executing non-coverage-increasing tests can significantly improve fuzzing efficiency. The key challenge is how to make this determination at a low cost effectively. Indeed, if we can decide whether or not a test increases coverage at a lower cost than a full test execution, we can reduce execution overhead and thus improve fuzzing efficiency.

KEY IDEA Determining if an input increases code coverage *without actually executing it* is beneficial but a paradoxical challenge. We introduce the notion of *prefix-guided execution* (PGE) to tackle this challenge. PGE leverages two key observations: (1) Only a tiny fraction of the mutated inputs increase coverage, thus requiring full execution, and (2) whether an input increases coverage may be accurately inferred from its partial execution. PGE monitors the execution of an input and applies early termination when the execution prefix indicates that the input is unlikely to increase coverage.

Generally, a program execution is a temporal transition sequence of program states. Code coverage metrics abstract over both program states and transitions. We refer to *full execution* as the complete temporal sequence, while *execution prefix* as the contiguous subsequence from the program entry. PGE is based on the hypothesis that whether a full execution leads to increased coverage can be effectively inferred from its prefix. Suppose that a short *execution prefix* suffices for accurately inferring whether the corresponding *full execution* increases code coverage, we can speed up fuzzing by focusing on only fully executing those coverage-increasing tests.

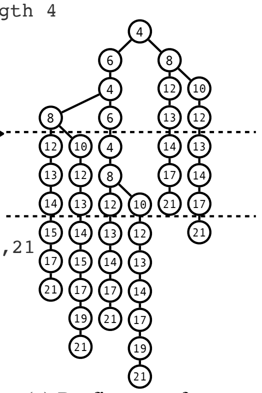
Example. Figure 1.2 illustrates the intuition behind PGE with a constructed code snippet. Figure 1.2(a) shows the constructed function `foo`. To maximize code coverage, *e.g.*, line coverage, a fuzzer needs to generate tests to reach lines `6`, `10`, `13`, `15`, and `19`, all of which rely on earlier program executions. For instance, reaching line `19` requires the satisfaction of three guards and the referred variables being suitably used/updated by previous statements.

```
1  int foo(int a, int b)
2  {
3    int ret = 0;
4    while ( a < 1)
5    {
6      a++, ret++;
7    }
8    if (b == 1)
9    {
10     a--;
11   }
12   if (a + b >= 0) {
13     ret++;
14     if (ret == 2)
15       ret++;
16   }
17   if (a==0 && b==1
18       && ret==3) {
19     ret = 0;
20   }
21   return ret;
22 }
```

(a) A constructed code snippet.

```
                  Prefix of length 4
foo(1, 0)  ⇒ 4, 8,12,13,14,17,21
foo(1, 1)  ⇒ 4, 8,10,12,13,14,17,21
foo(1, -1) ⇒ 4, 8,12,13,14,17,21
foo(0, 0)  ⇒ 4, 6, 4, 8,12,13,14,15,17,21
foo(0, -1) ⇒ 4, 6, 4, 8,12,13,14,15,17,21
foo(-1, 1) ⇒ 4, 6, 4, 6, 4, 8,10,12,13,14,17,19,21
foo(0, 1)  ⇒ 4, 6, 4, 8,10,12,13,14,15,17,19,21
foo(-1, 0) ⇒ 4, 6, 4, 6, 4, 8,12,13,14,17,21
foo(-1, -1)⇒ 4, 6, 4, 6, 4, 8,12,13,14,17,21
```

(b) Execution traces of inputs mutated from `foo(1, 0)`.



(c) Prefix tree of traces.

Figure 1.2: Motivation example for the intuition of PGE.

Figure 1.2(b) lists execution traces of tests mutated from the seed `(1,0)` in a top-down manner. Rectangular frames highlight coverage-increasing tests. Out of the eight mutated tests, three increased line coverage. We also build the corresponding prefix tree for these traces in Figure 1.2(c). From this prefix tree, we can see that, to differentiate all *full execution* traces, prefixes of length 7 are needed (indicated by the second dotted line). To separate coarser-grained coverage-increasing traces, prefixes of length at most 4 are then sufficient (indicated by the first dotted line). Figure 1.2(b) highlights in grey the first occurrences of unique prefixes of length 4. We can find that these unique prefixes cover all interesting tests, namely the seed and coverage-increasing tests. Suppose that a fuzzer monitors execution prefixes (with a fixed length of 4) and terminates an execution immediately whenever its prefix has occurred before. Fuzzing `foo` with tests in Figure 1.2(b) would result in 4 full executions on prefix-unique tests and 5 partial executions on the rest. The execution overhead is thus reduced on 5 out of the 9 tests. In practice, a large fraction of tests generated by fuzzers are neither coverage-increasing nor prefix-interesting. Guiding executions with such prefixes in fuzzing can potentially lead to significant cost reduction.

IMPLEMENTATION To demonstrate the potential of PGE, we implement a prototype on top of AFL++, which we call AFL++-PGE. We evaluate AFL++-PGE on Magma [52], a ground-truth benchmark set that consists of 21 programs from nine popular real-world projects. Our results show that, after 48 hours of fuzzing, AFL++-PGE finds more bugs, discovers bugs faster, and achieves higher coverage. Prefix-guided execution is general and can benefit the AFL-based family of fuzzers. The artifact for PGE, including all source code and data, is permanently available [74].

1.2.3 *Contribution:* SAND

Sanitizers provide robust test oracles for fuzzing to detect various security vulnerabilities related to undefined behavior effectively. Fuzzing on sanitizer-enabled programs has been the best practice to find software bugs. Since sanitizers need to instrument a target program to insert runtime checks heavily, sanitizer-enabled programs have much higher overhead compared to normally built programs.

```
1  _TIFFfree(*read_ptr);
2  ...
3  read_buff = *read_ptr;
4  if (!read_buff) {
5      read_buff = limitMalloc(buffsize);
6  }
7  else {
8      if (prev_readsize < buffsize) {
9          new_buff = _TIFFrealloc(read_buff, buffsize);
10         if (!new_buff) {
11             free(read_buff);
12             read_buff = limitMalloc(buffsize);
13         }
14         else
15             read_buff = new_buff;
16     }
17 }
18 read_buff[buffsize] = 0;
```

Figure 1.3: A simplified Use-after-Free bug from CVE-2023-26965. Line 18 triggers it as "read_buff" is never reallocated after being freed in line 1.

KEY IDEA Since sanitizers provide the security oracle for execution, all current fuzzers execute sanitizer-enabled programs on every fuzzer-generated input to verify validity. We now raise this question: *Can we decide whether an input triggers a bug without truly executing a sanitizer-enabled program?* Theoretically, it seems paradoxical and infeasible as only by executing an input can we know if the input is bug-triggering. However, our empirical evaluation will substantiate its feasibility. The key insight is that bugs are strongly connected to execution paths. For instance, Figure 1.3 shows a simplified code snippet from CVE-2023-26965, which contains a Use-after-Free bug in line 18. Normal and most execution paths are $\{1 \rightarrow 3 \rightarrow 5 \rightarrow 18\}$, $\{1 \rightarrow 3 \rightarrow 9 \rightarrow 11 \rightarrow 12 \rightarrow 18\}$, or $\{1 \rightarrow 3 \rightarrow 9 \rightarrow 15 \rightarrow 18\}$, where the freed buffer `read_buff` in line 1 is correctly re-allocated. However, when the execution path is $\{1 \rightarrow 3 \rightarrow 18\}$, the freed buffer `read_buff` is incorrectly used in line 18. This buggy execution has a unique path not seen in other normal executions.

Since triggering such control-flow sensitive bugs requires exercising unique execution paths, our intuition is that we can encapsulate inputs with unique execution paths by executing them on normally built programs, then only feed these inputs into sanitizer-enabled programs to reduce overall sanitization overhead. Conceptually, our intuition can effectively tackle control-flow sensitive bugs, e.g., Use-after-Free bugs, since triggering such kinds of bugs needs to exercise a unique execution path, e.g., from the free point to the use point. For other bugs, such as Buffer-Overflow and Use-of-Uninitialized-Memory, they are more "data sensitive". For example, triggering a Buffer-Overflow bug often requires significantly changing the buffer offset value. Nevertheless, such data-flow changes can often result from or be reflected by control-flow information [54]. As our illustration in Chapter 4 will show, Buffer-Overflow bugs are often control-flow related, such as unusual loop iterations, and thus can be identified via unique execution paths. In fact, previous studies [63, 145] have also implicitly shown that bugs correlate highly to executions.

Based on the above analysis, we present SAND, a new fuzzing framework that decouples sanitization from the fuzzing loop. SAND performs fuzzing on a *normally built program* and only invokes sanitizer-enabled programs when input is shown to be interesting. Since most of the generated inputs are not interesting, *i.e.*, not bug-triggering, SAND allows most of the fuzzing time to be spent on the normally built program. To identify interesting inputs, we introduce *execution pattern* for a practical execution analysis on the normally built program. Note that the PGE introduced before works at the early execution stage while SAND works after the execution has been completed. Thus, these two approaches are orthogonal and can, in principle, be used together.

IMPLEMENTATION We realize SAND on top of AFL++ and evaluate it on 12 real-world programs. Our extensive evaluation highlights its effectiveness: on a period of 24 hours, compared to fuzzing on ASan/UBSan-enabled and MSan-enabled programs, SAND respectively achieves 2.6x and 15x throughput and detects 51% and 242% more bugs.

1.3 RELIABILITY OF CODE ANALYSIS

To improve the reliability of our code, we typically adopt code analysis tools, such as bug detectors, to analyze and report issues in our code. For example, in the above-mentioned fuzzing process, whether or not the execution of an input results in a crash relies on the underlying bug detectors. That is, fuzzers only generate inputs, while the underlying detectors are the key to telling the fuzzer if an input triggers a bug or not. One of the most widely

```
 1 struct a { int x };
 2 struct a b[2];
 3 struct a *c=b, *d=b;
 4 int k = 0;
 5 int main() {
 6   *c = *b;
 7   k = 2;
 8   *c = *(d+k);
 9   return c->x;
10 }
```

```
(command line)
 $ gcc -O0 -fsanitize=address a.c
 $ ./a.out
 ==1==ERROR: AddressSanitizer:
 stack-buffer-overflow in a.c:8
 $
```

```
(command line)
 $ gcc -O2 -fsanitize=address a.c
 $ ./a.out
 $
```

Figure 1.4: Line 8 in `a.c` contains a stack-buffer-overflow (left). GCC ASan at -O0 successfully detects it (top right). GCC's Asan at -O2, however, overlooks it (bottom right). [43]

used detectors for undefined behaviors is sanitizers. While substantial research and engineering efforts have been made toward devising efficient fuzzers [105, 106] and reducing sanitizer costs [62, 163, 165], the robustness and reliability of sanitizers — essential for detection effectiveness — have received little attention from both academia and industry. In this part, we focus on the reliability of sanitizers.

Both GCC and LLVM, the two most popular C/C++ compilers, support sanitizers. Over the past five years, there were only 29 bug reports related to sanitizer correctness in the bug trackers of GCC and LLVM. Most of these reports (66%) were false positive issues, where sanitizers did not miss UBs but instead incorrectly reported correct executions as containing UB. False positive issues are indeed easy to be noticed in practice. For example, typical compiler testing work [66, 85, 159] involves generating valid programs as input, which can be trivially adapted to identify false positive issues. Conversely, false negative bugs in sanitizers typically result in a UB being missed and are thus difficult to observe.

Figure 1.4 illustrates a code snippet that triggers a false negative bug in GCC ASan. Since `d` points to the starting location of `b[2]`, the dereference `*(d+k)` at line 8 will cause a stack-buffer-overflow. When we compile and run this code with GCC ASan at *-O0*, ASan crashes the execution and generates a report as expected (Figure 1.4 top right). However, at *-O2*, it unexpectedly misses this UB (Figure 1.4 bottom right). This is a false negative bug of GCC ASan. As a UB detection tool, false negative bugs in sanitizers lead to missing UBs, thereby significantly impeding their effectiveness.

1.3.1 *Contribution:* UBFUZZ

In this thesis, we introduce the first effective testing framework for finding false negative (FN) bugs in sanitizers. The sanitizer bug shown in Figure 1.4 was discovered by our new framework.

KEY IDEA At a high level, the general workflow of our new framework is (1) generating a UB program, *i.e.*, a program exhibiting undefined behavior, and (2) compiling it with sanitizers and executing the compiled binary. If no sanitizer report on a UB program is produced, a potential sanitizer bug is detected. However, there are two main challenges in designing such a testing framework.

Challenge 1. The UB programs used for testing sanitizers need to be (1) diverse, *i.e.*, exercising different types of UB; (2) complex, *i.e.*, containing non-trivial syntax and semantics; and (3) easy-to-reduce, *i.e.*, can be easily reduced into a small problem suitable for reporting to sanitizers' developers. Unfortunately, there is no existing program generator that can produce such programs. To tackle this challenge, we introduce *Shadow Statement Insertion*, a general and effective approach for introducing UB into a valid seed program. The generated UB programs are subsequently utilized for differential testing of multiple sanitizer implementations.

Challenge 2. As has been discussed in COMPDIFF, given an input UB program, compilers may optimize away the UB in the program. Sanitizers are implemented as a component of a compiler. The actual programs that sanitizers take as input have been optimized by the compiler. Thus, missed reports can also be due to aggressive compiler optimizations instead of sanitizer FN bugs. To tackle this challenge, we introduce a novel test oracle, namely *crash-site mapping*, to accurately discern whether a discrepant sanitizer report stems from sanitizer FN bugs or compiler optimizations.

IMPLEMENTATION We have incorporated our techniques into UBFUZZ, a practical tool for testing sanitizers. Over a five-month testing period, UBFUZZ successfully found 31 bugs in both GCC and LLVM sanitizers. These bugs reveal the serious false negative problems in sanitizers, where certain UBs in programs went unreported. This research paves the way for further investigation in this crucial area of study. The artifact for UBFUZZ, including all source code and data, is permanently available [75].

1.4 RELIABILITY OF CODE COMPILATION

Suppose you have written a piece of perfect code that is fully correct, does that mean your software is fully correct? Or, in other words, *is source code correctness equivalent to software correctness?* Compilers help us translate the code written by us in a high-level programming language into low-level machine code that can be run on different platforms. In this thesis, we show that due to compiler bugs, the binary or low-level machine code can be buggy even if the source code is correct.

1.4.1 *Contribution:* CREAL

To find compiler bugs, the best present practice is to design a generator to randomly produce a large number of valid programs. Existing generation- and mutation-based generators introduce a fundamental limitation: They are rule-based generators and, thus, are unable to produce programs with features beyond the underlying rules. These approaches rely on *predefined syntactical and semantic rules* in constructing programs. For example, to avoid undefined behavior, Csmith resorts to heavy-handed safe wrappers, which limits the expressiveness of the generated programs [31]; to guarantee termination, Csmith only generates constant-bounded loops, meaning that all loop conditions are constant; EMI-based mutators [67, 139] augment generator-produced programs with random code and thus inherit the limitations. Due to these constraints, experience shows that all random program generators will eventually saturate and find very few bugs [2]. Extending the scope of current approaches by integrating more rules is theoretically feasible but is challenging and labor-intensive in practice.

KEY IDEA We propose using code from real-world projects for compiler testing. The core idea is first to extract code snippets from real-world projects and then inject them into a seed program to enrich its code features. Our key observation is that real-world code exercises a diverse range of syntactical and semantic code features, which are often hard for random program generators or mutators to support. Figure 1.5a shows a loop from

```
1 src = data + offsetp;
2 do {
3     b = *src++;
4     ret |= ((b & 0x7f) << shift);
5     (*offsetp)++;
6     shift += 7;
7 } while ((b & 0x80) != 0);
```

a. A loop from the FreeBSD project.

```
for (p_1 = 0; p_1 <= 6; p_1 -= 8) {
    int32_t l_6 = 0x2E2B9C6;
    l_3 ^= (p_1 > (l_5[0] ^ l_6));
} // Csmith

for(i0=var1;i0 < arr2[0]%var3; i0+=1){
    var4 ^=  arr5[i6] << (var3+1);
} // YarpGen v.2
```

b. Loops taken from Csmith and YARPGen v.2.

Figure 1.5: Loops from real-world project, Csmith, and YARPGen v.2.

FreeBSD [1], a UNIX-like OS written in C. This code contains an unbounded `while` loop, where the loop index "`b`" is modifiable at both lines 3 and 5. This code helped us find a latent miscompilation bug in LLVM. Figure 1.5b shows two common loops from Csmith and YARPGen v.2, respectively. The first loop is from Csmith. It is constant-bounded, which is true for all Csmith-produced loops. The second loop is from YARPGen v.2 and has more complex loop indexes such as `i0` and `arr2`. However, unlike Figure 1.5a, all these variables are not modifiable in the loop body. Generators need to capture code semantics at generation time to generate well-formed programs, which is hard for complex loops.

To use real-world code, we begin by extracting well-formed functions from real-world projects. Next, we inject function calls into seed programs to establish vigorous connections between real-world functions and the seed program. Our code injection technique ensures that all produced programs are well-formed. There are three main benefits of using real-world code at the function level: (1) most compiler optimizations work at the function level [89, 41], ensuring real-world functions can already exercise rich features, (2) it is relatively painless to guarantee the validity of extracted functions in terms of undefined behavior, and (3) it requires moderate changes to the seed program, making it uncomplicated to maintain the validity of the resulting program. An alternative technique is to inject code chunks directly into seed programs. However, this requires extensive modifications to both the extracted real-world code and the seed programs, which can easily result in invalid programs. Nonetheless, we consider this orthogonal approach to be an interesting future work. Our evaluation will show that using real-world functions as building blocks can effectively

1 https://github.com/freebsd/freebsd-src/blob/main/contrib/elftoolchain/libdwarf/libdwarf_rw.c

explore various code features. Note that the expressiveness of our approach depends on both the seed programs and the function database. Therefore, (1) Creal does not replace existing program generators but rather complements them by boosting their expressiveness. Generators like Csmith [159], YARPGen [86], or EMI-based tools [66] can all be used to provide the seed programs; (2) since the function database has a limited size, our approach will eventually saturate until new functions or extensions are developed.

IMPLEMENTATION We implement our idea in a tool named Creal. Since our approach is data-hungry, we need a diverse range of functions from real-world projects to fuel Creal. We develop a function extractor, with which we construct a function database containing more than 51,000 real-world functions from 146 popular open-source projects, such as Git, Linux Kernel, and OpenSSL, to name a few. With this database, we evaluate Creal by stress-testing the latest versions of C compilers, namely GCC and LLVM. In a nine-month testing period, our tool identified 132 compiler bugs. Of those, 121 were confirmed, and 97 were fixed by the compiler developers. It is worth noting that most of the bugs were miscompilations, which are the most harmful and difficult-to-detect compiler bugs. The artifact for Creal, including all source code and data, is permanently available [73].

Part I

RELIABILITY OF CODE

2

FINDING UNSTABLE CODE VIA COMPILER-DRIVEN DIFFERENTIAL TESTING

In this chapter, we focus on detecting undefined behaviors in C/C++ programs. As has been discussed in Chapter 1, compilers can assume that undefined behavior will never occur in the input program, which allows many optimization opportunities. A consequence of such an assumption is for code that contains undefined behavior, different compiler implementations may generate semantically different binaries. Previous study [151, 152] has shown that undefined behaviors may cause optimization-unstable code, code that could be unexpectedly discarded by compiler optimizations. In this chapter, we refer to code that has inconsistent semantics across compiler implementations due to undefined behavior as *unstable code*. We introduce compiler-driven differential testing (CompDiff), a simple yet effective approach for finding unstable code in C/C++ programs.

The work in this chapter was published in [77].

Example of unstable code. Listing 1 shows an example of unstable code, where the `if` guard in line 9 tries to handle possible integer overflow. But `offset+len` can never be less than `offset` unless undefined behavior, *i.e.*, signed integer overflow, occurs. According to the C17 standard [14], compilers can do arbitrary optimizations with the assumption that undefined behavior never occurs. The consequence is that an optimizing compiler (*e.g.*, `clang-O2`) optimizes away the second `if` branch (lines 9-11) while a less optimizing compiler (*e.g.*, `clang-O0`) keeps it. That means the compiled binaries have different semantics. For example, if we call `dump_data(INT_MAX-100, 101)`, the optimized binary will dump the buffer starting from `data+INT_MAX-100` and return 0, while the unoptimized binary will dump nothing and return -1. On one hand, this issue leads to a security hole in the optimized binary, as a large range of illegal memory data could be dumped. On the other hand, it breaks the functional correctness of the code as its binaries compiled by different compilers may produce divergent outputs.

Key idea. From the above example, we can observe that unstable code leads to different execution semantics across compilations once undefined

```
1  /* dump a chunk of buffer*/
2  int dump_data (int offset, int len) {
3      char *data = /* buffer head */;
4      int size = /* size of buffer*/;
5      if (offset + len > size ||
6          offset < 0 || len < 0) {
7          return -1;
8      }
9      if (offset + len < offset) {
10         return -1;
11     }
12     /* dump from data+offset
13        to data+offset+len */
14     dump(data+offset, len);
15     return 0;
16 }
```

Listing 1: The second if guard in line 9 got optimized away by clang because it would only be evaluated to true when a signed integer overflow happened.

behavior has been triggered. This conforms to the standard that compilers can, in principle, do arbitrary optimizations on such erroneous code. In this thesis, we make use of this fact and propose a simple, straightforward, yet effective approach for finding unstable code. Our approach involves three steps. First, compile the target program with different compiler implementations to get a set of binaries. We consider different compilers and optimizations as different compiler implementations. For example, `gcc-O0`, `gcc-O2`, and `clang-O2` are three different compiler implementations. Second, run these binaries on the same set of test inputs and collect their outputs. Finally, compare outputs produced by different binaries on the same input and report discrepancies. For a deterministic program, i.e., repeated executions of the program on the same input always yield the same output, discrepant outputs on the same input imply the presence of unstable code or undefined behavior. We call our approach *compiler-driven differential testing* (CompDiff). For the example in Listing 1, CompDiff can successfully detect the issue because of the divergent outputs on the same input. For non-deterministic or multi-threaded programs, they may have non-deterministic internal execution traces. But as long as they have deterministic output, they can be analyzed with CompDiff. Note that, CompDiff assumes compiler implementations are bug-free. Compiler bugs,

Table 2.1: Scopes of sanitizers and CompDiff.

Approach	Scope
ASan	Memory errors (*e.g.* buffer-overflow)
UBSan	Miscellaneous UBs (*e.g.* division-by-zero)
MSan	Use of uninitialized memories.
CompDiff	A diverse range of UBs.

which are rare for mature compilers [101], may indeed cause divergent outputs and thus be caught by CompDiff. As will be shown in our evaluation, compiler bugs are rarely encountered in real-world software. Once it happens, developers are willing to diagnose and report it.

CompDiff's design also covers bugs that are not due to undefined behavior. As long as a bug results in output discrepancy across compiler implementations, it will be detected by CompDiff. Our evaluation will show that CompDiff indeed finds real bugs that are not undefined behavior. We consider this as an additional benefit of CompDiff.

Existing work. Industry and academics have proposed a plethora of static and dynamic tools for finding frequently occurring undefined behaviors such as buffer overflow, integer overflow, division by zero, *etc.* Static tools [152, 104, 25] analyze source code without executing it to detect certain types of errors. They typically build upon heuristics and suffer from both false positives and false negatives. Dynamic tools [13, 107], on the contrary, perform analysis of concrete executions and normally incur no false positives. Sanitizers, such as AddressSanitizer (ASan) [130], UndefinedBehaviorSanitizer (UBSan) [88], and MemorySanitizer (MSan) [136], are widely used in practice. They insert checks into necessary program locations to detect undefined behaviors at run-time. For the example in Listing 1, UBSan would insert a check around each `offset+len` to verify whether or not its value exceeds `INT_MAX`. Thanks to their strong bug detection ability, sanitizers have become de facto state-of-the-art for discovering UBs, especially in fuzzing.

Sanitizers cover many frequently occurring UBs. Each sanitizer is designed for certain classes of UBs. Table 2.1 lists scopes of three widely used sanitizers and our CompDiff. On the one hand, CompDiff covers a broader range of UB even than the combination of these sanitizers. The reason is that our design stems from unstable code, a common consequence of UB. Sanitizers contrarily design customized checks for each kind of UB. Since not all UBs have applicable checks, some UBs cannot be detected by sanitizers. We will show in Section 2.1 three examples where sanitizers fail to detect them. On the other hand, sanitizers have high bug coverage for UBs, which they specialize in. CompDiff, however, may miss many of them. We argue that CompDiff is not to replace sanitizers but to complement them by covering extra UBs.

To improve CompDiff's practicality and detect unstable code in real-world software, we integrate CompDiff into AFL++ [34], the most widely-used general-purpose fuzzer. Our evaluation shows that, on the Juliet benchmark tests, CompDiff uniquely identified 1,409 bugs that sanitizers failed to. On 23 popular open-source C/C++ projects, CompDiff-AFL++ discovered 78 new bugs, 66 of which were confirmed, and 52 were fixed by the developers. Of these new bugs, 36 were not detected by sanitizers. Our evaluation also confirms the strong detection ability of sanitizers on certain classes of bugs.

Main contributions. In summary, the main contributions are summarized as follows:

- We propose CompDiff, a simple, straightforward, yet effective approach for finding unstable code.
- We integrate CompDiff into the popular fuzzer AFL++.
- We evaluate CompDiff on both benchmark and real-world programs. The results show that CompDiff significantly complements sanitizers.

The artifact for CompDiff, including all source code and data, is permanently available [72].

2.1 ILLUSTRATIVE EXAMPLES

This section illustrates three real-world examples, which we use to demonstrate how CompDiff enables the discovery of unstable code and why sanitizers fail to detect them.

Example 1: Invalid pointer comparison. Listing 2 shows a piece of unstable code found by CompDiff in Binutils. The pointer comparison in line 5

```
1 int display_debug_frames (...) {
2     char *saved_start=/*point to object A*/;
3     char *look_for =/*point to object B*/;
4     ...
5     if (look_for <= saved_start) {...}
6     else {...}
7 }
```

Listing 2: A pointer comparison UB found by CompDiff in binutils/dwarf.c. The if check in line 5 is evaluated differently (either true or false) across compiler implementations (`https://sourceware.org/bugzilla/show_bug.cgi?id=27836`).

is UB as `look_for` and `saved_start` are pointing to different objects. The standard [14] (§6.5.8) describes that such undefined behavior happens when *"Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators."* None of the sanitizers can detect this kind of UB because it remains unknown how to design a proper check for it. CompDiff can easily detect this issue because the `if` guard will be evaluated differently across compiler implementations, and thus divergent outputs will be observed.

Example 2: Evaluation order of subexpressions with conflict side effects. Listing 3 shows an example of unstable code in the well-known network packet analyzer Tcpdump [140]. The function `ND_PRINT` (line 9) dumps formatted network information. In this code snippet, developers try to dump fields `p1` (line 10) and `p2` (line 11) by calling the function `GET_LINKADDR_STRING` twice. These two calls are also arguments to the function `ND_PRINT`. According to the standard, compilers can evaluate function arguments in any order. However, *"If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings."* [14] (§6.5.0). This example matches the definition of this undefined behavior and becomes unstable. First, the function `GET_LINKADDR_STRING` uses a static char array `buffer` to store the resulting string. The memory region pointed to by `buffer` will be shared across function calls. Since there are two calls to this function, the result of the first call, which is stored in `buffer`, will be overwritten by the second call. Thus in the dumped string, the two fields `who-is` and `tell` will always be the same. Second, since the language specification poses no restriction on the evaluation order of function arguments, different compil-

```
char * GET_LINKADDR_STRING(int8_t *p) {
    static char buffer[BUF_SIZE];
    /* write *p to buffer */
    ...
    return buffer;
}
void arp_print(...) {
    ...
    ND_PRINT("who-is %s tell %s",
        GET_LINKADDR_STRING(p1),
        GET_LINKADDR_STRING(p2));
    ...
}
```

Listing 3: An example of unstable code simplified from tcpdump/print-arp.c. The two calls to `GET_LINKADDR_STRING` are arguments to the function `ND_PRINT` (`https://github.com/the-tcpdump-group/tcpdump/issues/919`).

ers may evaluate these two `GET_LINKADDR_STRING` calls in a different order. If we compile Tcpdump with `gcc` and `clang` separately, the obtained two binaries will evaluate the arguments of `ND_PRINT` in reverse order, leading to inconsistent dump strings. Specifically, `clang` evaluates the arguments from the first to the last, *i.e.*, `p2` will be dumped to both `who-is` and `tell`; while `gcc` evaluates the arguments from the last to the first, *i.e.*, `p1` will be dumped to both attributes.

To discover this issue, we need to have at least two compiled tcpdumps from `gcc` and `clang`, respectively, and tests that can reach the unstable program location. We identified this issue with our CompDiff-AFL++ tool, where CompDiff was configured to compile a target with multiple compiler implementations from both `gcc` and `clang`. The back-end AFL++ generated tests that reached the target location.

All sanitizers currently do not support the detection of this type of issue. Extending sanitizers to support such detection requires the design of a new checker that could examine whether or not multiple subexpressions have side effects on conflict memory regions. It remains unknown how to implement such a checker.

Example 3: Uninitialized memory usage. Listing 4 shows a piece of unstable code due to the use of an uninitialized variable. The developers might think that although the variable `l` is uninitialized, its initial random value

```
1 std::ostream& CanonMakerNote::print0x000c(
2     std::ostream& os, const Value& value) {
3
4     std::istringstream is(value.toString());
5     uint32_t l;
6     is >> l;
7     return os << std::hex
8               << ((l & 0xffff0000) >> 16);
9 }
```

Listing 4: A use of uninitialized variable in exiv2 where l stays uninitialized even after line 6 when is is an empty string (https://github.com/Exiv2/exiv2/issues/1717).

should be overwritten in line 6 with the content in is. However, in a corner case where is is an empty string the variable l will remain unchanged. The uninitialized value will then be used for the rest of the execution, in this case printing out to ostream. As the value of the uninitialized variable is indeterminate [14] (§6.7.9) and depends on run-time memory layout, different compilers and optimizations may allocate different values to it.

MemorySanitizer supports the detection of uninitialized memory usage, where uninitialized values have to be used to determine code branches, *e.g.*, an if guard relies on an uninitialized value. To avoid false positives, it does not support cases such as the one shown in the example.

CompDiff-AFL++ can detect this issue because 1) the back-end AFL++ can generate tests that cause the variable is to be empty and thus l to be different across binaries; 2) CompDiff captures divergent outputs.

Limitations. Although CompDiff can cover extra bugs than sanitizers, it cannot detect as many issues as sanitizers for certain kinds of UBs. The reasons are twofold. First, some UBs do not lead to unstable code. Although compilers in theory could generate arbitrary binaries for code having UB, they in practice may not exploit the UB or generate semantically equivalent binaries. Second, Even if a program contains unstable code, the erroneous behavior may not propagate to the final outputs.

2.2 APPROACH

This section details our approach to finding unstable code. Section 2.2.1 formalizes unstable code and presents our proposed compiler-driven differential testing for the detection of unstable code. Section 2.2.2 demonstrates the implementation details of integrating CompDiff into AFL++.

2.2.1 *Compiler-Driven Differential Testing*

Both C [14] and C++ [15] standards pose no requirements on how a compiler behaves on code fragments that invoke undefined behavior. A compiler implementation *w.r.t.* the programming language specification can thus in principle do arbitrary transformations and optimizations on such erroneous code fragments. Intuitively, for a program with deterministic output, *i.e.*, repeated executions of the program on the same input always yield the same output, if the binaries compiled by any two legal and correct compiler implementations produce different outputs on some input, there has erroneous code fragment in the program. We call such erroneous code fragments leading to divergent outputs across compilations *unstable code*.

To formalize unstable code, let $\mathcal{C}_a$ and $\mathcal{C}_b$ be *any* two legal compiler implementations. For a program $\mathcal{P}$ with deterministic output, $\mathcal{C}_a$ and $\mathcal{C}_b$ compile program $\mathcal{P}$ to binaries $\mathcal{B}_a$ and $\mathcal{B}_b$, respectively. With these notations, we introduce unstable code from a dynamic testing perspective as follows:

Definition 1 (Unstable Code)**.** *$\mathcal{P}$ has unstable code if there exists an input such that executing $\mathcal{B}_a$ and $\mathcal{B}_b$ on the input produces different outputs.*

This definition shows that enumerating inputs on binaries compiled by all possible compiler implementations may help us to find unstable code. Based on this intuition, we propose compiler-driven differential testing (CompDiff) for detecting unstable code in real-world programs. For a program $\mathcal{P}$, the general workflow of CompDiff is as follows:

1) Find a set of legal compiler implementations $\mathcal{C}_i$, $i \in [1, 2, \cdots, k]$.

2) Compile $\mathcal{P}$ with each $\mathcal{C}_i$ to obtain binaries $\mathcal{B}_i$, $i \in [1, 2, \cdots, k]$.

3) Find an input set $\mathcal{I}$ for $\mathcal{P}$.

4) For each input $t \in \mathcal{I}$, run each $\mathcal{B}_i$ on it and obtain outputs o_i. If there exists $i, j \in [1, \cdots, k]$ and $i \neq j$ such that $o_i \neq o_j$, report t as bug-triggering input.

The above workflow shows three key factors in CompDiff, *i.e.*, compiler implementations, input set, and output examination. In the following, we will discuss each of these factors in detail.

Compiler implementations. Ideally, we could find only two compiler implementations that always behave differently on unstable code. However, such ideal compilers do not exist. In practice, we should utilize a set of concrete compiler implementations. For a given target, there are typically plenty of metamorphic compiler implementations. For instance, `clang` has hundreds of optimization passes available for developers to choose from, which results in a combinatorial explosion of possible compiler implementations. Popular compilers, such as gcc [42] and clang [90], feature well-engineered optimization levels (*i.e.*, `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`) for users to choose from. For open-source C/C++ projects, developers often use different compilers and optimization levels when developing, testing, and releasing code. Users may also freely choose their own preferences when compiling code. These compilers and optimization levels offer us a good collection of compiler implementations for CompDiff. The reason is twofold. First, each of these compiler implementations uses a different set of transformations and optimizations, which increase the possibility of exploiting unstable code across them. Second, all of these compiler implementations are widely used in real-world cases by either developers or users. As long as CompDiff discovers a discrepancy, the issue is likely affecting real users and thus persuading.

Input set. Since CompDiff is a dynamic testing approach, concrete execution is required to detect bugs. One can use a test suite provided by developers or generated with existing test generation tools. Fuzzing is one of the most popular automated testing techniques for discovering software defects due to its simplicity and effectiveness. By generating a tremendous amount of mutated inputs with feedback guidance, the fuzzer is powerful in finding interesting program paths. In this thesis, we use a fuzzer as the test generation tool to power CompDiff. Section 2.2.2 will demonstrate how we integrate CompDiff into AFL++ for finding unstable code in real-world software. Note that CompDiff has no particular requirements on the underlying fuzzer and is generally applicable to other fuzzers.

Output examination. Conceptually, our definition of unstable code shows that finding them in a program requires verifying the semantic equivalence of its compiled binaries. Full verification is infeasible due to its complexity and undecidability [28, 125]. To practically reason about semantic equivalence, our design only concerns final outputs of binaries on a concrete input. There are two reasons behind our choice. First, it is easy and cost-efficient to obtain input/output pairs in practice. Unlike other techniques like sanitizers, such design requires no modification or instrumentation to the target

Algorithm 1: CompDiff-AFL++

```
Input: Seed pool $\mathcal{S}$.
1  while $\neg$Abort() do
2      $s \leftarrow$ SelectSeed($\mathcal{S}$)
3      $s' \leftarrow$ Mutate($s$)
4      _ $\leftarrow$ Execution($s', \mathcal{B}_{fuzz}$)
5      if $s'$ causes failure on $\mathcal{B}_{fuzz}$ then
6          save $s'$ to disk
7      if $s'$ increases coverage on $\mathcal{B}_{fuzz}$ then
8          add $s'$ to $\mathcal{S}$
       /* CompDiff: Run $s'$ on each $\mathcal{B}_i$ and examine output
          consistency.                                              */
9      for $i = 1$ to $k$ do
10         $o_i \leftarrow$ Execution($s', \mathcal{B}_i$)
11     if !$(o_1 = o_2 = \cdots = o_k)$ then
12         save $s'$ to disk
```

program. Second, it is convincing as proof of the presence of unstable code. Since all compiler implementations are used in the real world, any discrepancy in the final outputs indicates that the functional correctness of the program has been affected in at least one compiler implementation. Unfortunately, such a design will miss bugs when the erroneous state does not propagate to the final output. One may argue that instead of only examining final outputs, we can check programs' intermediate results, *e.g.*, return values from all functions, to extend CompDiff's capability. However, it 1) may incur many false positives due to compilers' optimizations such as inlining, 2) is less persuading than final outputs in motivating developers to fix the error, and 3) requires significant and non-trivial implementation efforts in obtaining intermediate results at run-time. It is, however, worth further exploration in the future.

2.2.2 CompDiff-*AFL*++

We here demonstrate integration details of CompDiff-AFL++. Since CompDiff is orthogonal to fuzzers, we hope our demonstration can guide the future adoption of CompDiff in other fuzzers.

In CompDiff-AFL++, there are multiple binaries compiled from the target program $\mathcal{P}$. The first one is $\mathcal{B}_{fuzz}$, which is compiled by the fuzzer-configured compiler $\mathcal{C}_{fuzz}$. The compiler $\mathcal{C}_{fuzz}$ injects instrumentation code into $\mathcal{P}$ such that the fuzzer can collect coverage feedback from the resulting binary $\mathcal{B}_{fuzz}$. If sanitizers are enabled, $\mathcal{C}_{fuzz}$ also insert sanitizer checks to $\mathcal{B}_{fuzz}$. Note that $\mathcal{B}_{fuzz}$ is compiled the same as in normal AFL++. The remaining binaries are all for CompDiff. As has been discussed in Section 2.2.1, we recommend the use of compilers `gcc` and `clang` with different optimization levels to form the set of compiler implementations $\mathcal{C}_i, i \in [1, \cdots, k]$. Each $\mathcal{C}_i$ compiles $\mathcal{P}$ to binary $\mathcal{B}_i$. We inject some lightweight instrumentation code into each $\mathcal{B}_i$ for efficient execution. We will discuss the instrumentation on $\mathcal{B}_i$ after introducing the algorithmic sketch of CompDiff-AFL++.

Algorithm 1 shows the high-level workflow of CompDiff-AFL++. The unhighlighted part describes AFL++'s main process:

1) (line 2) Select a seed input from the seed pool.
2) (line 3) Mutate the input with one of the available mutation operators.
3) (line 4) Execute the program on the mutated input and collect the feedback, such as code coverage from the execution.
4) (line 5-8) If the new input causes a crash, save it to disk; if it increases coverage, add it to the seed pool; otherwise, drop it. Then go to step 1).

The highlighted part in Algorithm 1 shows where and how CompDiff works in AFL++. It first runs the new input on each $\mathcal{B}_i$ (lines 9-10), then cross-checks their outputs and saves the input if a divergence is found (lines 11-12). The workflow illustrates that CompDiff does not interfere with AFL++'s normal procedures but only augments it with the extra test oracle provided by CompDiff. Thus, other fuzzing enhancements to AFL++ are compatible with CompDiff-AFL++. For example, sanitizers work by instrumenting $\mathcal{B}_{fuzz}$ to expose more bugs, and thus, they can be normally used in CompDiff-AFL++. Next, we will discuss key implementation details in CompDiff-AFL++.

Instrumentation on $\mathcal{B}_i$. Each compiler implementation $\mathcal{C}_i$ primarily specifies the used compiler and optimization level. For instance, in our default setting, the compiler $\mathcal{C}_1$ uses `CC=clang CXX=clang++ CFLAGS="-O0" CXXFLAGS="-O0"` and compiler $\mathcal{C}_2$ uses `CC=clang CXX=clang++ CFLAGS="-O1" CXXFLAGS="-O1"`. To reduce the burden of launching target binaries $\mathcal{B}_i$, we also inject *forkserver* [162] instrumentations into them. Forkserver has been widely used in many fuzzers for the same purpose. At a high level, when the fuzzer needs to execute an input on $\mathcal{B}_i$, it first writes the input to shared memory, then notifies the forkserver in $\mathcal{B}_i$. After the forkserver receives the notification, it forks itself, runs the input on the forked child process, and informs the fuzzer when it is done. Interested readers can find further details in [162].

Output examination. AFL++, by default, drops all outputs emitted from the binary. To obtain output from each $\mathcal{B}_i$, we redirect output, as well as error, from each $\mathcal{B}_i$ to a file using `dup2()`. We then compare the checksum values of these files to find discrepancies. We reuse the MurmurHash3 [3] hash function supported by AFL++ for the checksum.

Bug-triggering inputs. We save all inputs that triggered output discrepancies into a separate directory "diffs/" for future diagnosis. Similar to crash-triggering inputs in normal fuzzing, there are many inputs that trigger the same bug. It is non-trivial to automatically identify unique discrepancies, especially in the context of differential testing. We currently rely on manual analysis of reported discrepancies to triage bug reports. Automated triage, as well as debugging, are discussed further in Section 2.4.

2.3 EVALUATION

In our evaluation, we use `gcc 11.1.0` and `clang 13.0.1`, the latest stable versions at the beginning of our evaluation, as the back-end compilers in CompDiff and CompDiff-AFL++. These two compilers are selected because of their widespread adoption in open-source C/C++ projects. To thoroughly understand the capability of different optimization levels, we utilize all frequently used ones, *i.e.*, `-O0`, `-O1`, `-O2`, `-O3`, and `-Os` in both compilers. The combination gives us 10 different compiler implementations. Our CompDiff-AFL++ is implemented atop AFL++ version 3.15a. All experiments are done in a server equipped with an AMD Ryzen Threadripper 3990X 64-Core 2.9GHz CPU and 256 GB RAM, and run on Ubuntu 20.04.3 LTS.

We evaluate the effectiveness and practicality of CompDiff and CompDiff-AFL++. To understand the capability of CompDiff in finding unstable code, we evaluate it on a collection of benchmark programs from the Juliet test suite. These programs contain a diverse range of undefined behaviors, and the ground truth is available. We then use 23 well-maintained open-source C/C++ projects to evaluate the bug-finding ability of CompDiff-AFL++ in real-world software. There are a plethora of static and dynamic tools for detecting common UBs. We evaluate three popular and widely-used C/C++ static analyzers, *i.e.*, Coverity [127], Cppcheck [25], and Infer [104], on the Juliet test suite. These tools implement state-of-the-art techniques for detecting various program issues and were used in previous study [83, 60]. Since CompDiff is a dynamic analysis tool, we also compare it with sanitizers, the state-of-the-art dynamic analysis tools, on both the Juliet test suite and real-world software. Other dynamic tools such as Valgrind [107] and Dr.Memory [13] do not have better detection ability than sanitizers when source code is available [29]. Thus we compare CompDiff with sanitizers only. Specifically, we compare our tool with three widely-used sanitizers, *i.e.*, AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan), and MemorySanitizer (MSan).

2.3.1 *Effectiveness of* CompDiff *in Benchmark Programs*

The Juliet test suite C/C++ [109] released by NIST contains a collection of test cases, which are classified based on MITRE's Common Weakness Enumeration (CWE) classification system. This test suite has been widely used to evaluate both static analysis tools [83, 104] and dynamic testing approaches [29]. Each test case can run as an independent program and contains two variants: a *bad* variant that contains a flaw and a *good* that does not. All bad variants can be used to evaluate the bug detection rate of a tool while good variants can be used to evaluate the false positive rate of a tool.

Not all CWEs are due to undefined behaviors, many of which are insecure or unsafe operations such as hard-coded passwords (CWE-256). In order to evaluate the effectiveness of CompDiff in finding unstable code, we manually analyzed each CWE category and selected CWEs that represent bugs due to undefined behavior. Since we only deal with programs that have deterministic output, we excluded tests that deliberately change outputs per run. We also removed tests that timed out after 5 seconds [29]. This extraction gave us 18,142 tests spanning 20 CWEs. Table 2.2 shows an overview of the selected CWEs.

We analyze each test with Coverity, Cppcheck, Infer, ASan, UBSan, and MSan, in addition to our tool, CompDiff. For clear presentation, we merge

Table 2.2: Overview of selected CWEs.

CWE-ID	**Description**	**#Tests**
CWE-121	Stack Based Buffer Overflow	2,951
CWE-122	Heap Based Buffer Overflow	3,575
CWE-124	Buffer Underwrite	1,024
CWE-126	Buffer Overread	721
CWE-127	Buffer Underread	1,022
CWE-415	Double Free	820
CWE-416	Use After Free	394
CWE-475	Undefined Behavior for Input to API	18
CWE-588	Access Child of Non Struct. Pointer	80
CWE-590	Free Memory Not on Heap	2,280
CWE-685	Function Call With Incorrect #Args.	18
CWE-758	Undefined Behavior	523
CWE-190	Integer Overflow	1,564
CWE-191	Integer Underflow	1,169
CWE-369	Divide by Zero	437
CWE-476	NULL Pointer Dereference	306
CWE-680	Integer Overflow to Buffer Overflow	196
CWE-457	Use of Uninitialized Variable	928
CWE-665	Improper Initialization	98
CWE-469	Use of Pointer Sub. to Determine Size	18
Total		**18,142**

Table 2.3: Bug detection rates (%) and false positive rates (%) on the Juliet tests. The highest bug detection rates are highlighted in green. False positive rates of static tools are shown in columns FP. Since all sanitizers and CompDiff have no false positive on the Juliet tests, we omit their FP values. The last column (#Unique) lists the number of bugs that are uniquely detected by CompDiff compared to sanitizers.

CWE-IDs	**Description**	**Static Tools**						**Sanitizers**				**CompDiff**	
		Coverity	FP	Cppcheck	FP	Infer	FP	ASan	UBSan	MSan	**Total**	Detected	#Unique
121∼127 127, 415, 416, 590	Memory error	39%	46%	13%	6%	37%	21%	94%	✗	✗	94%	63%	137
475	UB for input to API	100%	0%	100%	0%	0%	0%	100%	✗	✗	100%	100%	0
588	Bad struct. pointer	32%	21%	0%	4%	2%	2%	49%	✗	✗	49%	99%	40
685	Bad function call	100%	0%	100%	0%	0%	0%	100%	✗	✗	100%	100%	0
758	UB	100%	4%	0%	35%	0%	0%	36%	✗	✗	36%	92%	293
190, 191, 680	Integer error	22%	21%	0%	2%	49%	25%	✗	33%	✗	33%	11%	31
369	Divide by zero	54%	23%	8%	3%	3%	7%	✗	54%	✗	54%	29%	5
476	Null pointer deref.	69%	9%	29%	3%	77%	69%	✗	92%	✗	92%	93%	3
457, 665	Uinitialized memory	44%	56%	24%	15%	9%	13%	✗	✗	7%	7%	92%	882
469	UB of pointer Sub.	0%	0%	0%	0%	0%	0%	✗	✗	✗	0%	100%	18

tests with similar causes. We use *bad* (buggy) variant of each test. We evaluate the bug detection rate and the false positive rate of each tool. Bug detection rate (or recall) is the percentage of all real bugs detected by the tool. False positive rate is the percentage of incorrect reports (or false alarms) out of all reports produced by a tool. Table 2.3 shows the bug detection rates (%) and false positive rates (%) of each tool. Since all sanitizers and CompDiff do not have false positive reports on the Juliet test suite, we omit their false positive columns for better presentation. The last column shows the number of bugs that can be uniquely discovered by CompDiff compared to sanitizers. We next discuss five findings on the results.

➤ **Finding 1: Static tools have non-negligible false positive rates and relatively lower bug detection rates compared to CompDiff.**

All static tools show non-negligible false positive rates. Coverity, Cppcheck, and Infer have 0% $\sim$ 46%, 0% $\sim$ 35%, and 0% $\sim$ 69% false positive rates, respectively. On the contrary, CompDiff has zero false positive rate. Cppcheck can detect the same number of bugs as sanitizers and CompDiff on CWE-475 and CWE-685. Infer detects the most number of bugs on CWE-190$\sim$680 while still having a 25% false positive rate. For the rest of the bug categories, CompDiff has significantly higher bug detection rates than Cppcheck and Infer. For example, on CWE-588, CompDiff detects 99% of bugs while Cppcheck and Infer only detect 0% and 2% of bugs, respectively. On most of the CWEs, CompDiff detects more bugs than Coverity. For bugs from "UB", "Integer error", and "Divide by zero", Coverity has higher bug detection rates. However, Coverity has 4% to 23% false positive rates on these bugs.

Overall, CompDiff shows stronger bug detection ability than static analysis tools. In practice, static and dynamic tools have complementary strengths and should be used together to maximize the bug detection rate.

➤ **Finding 2: CompDiff complements sanitizers by discovering many extra bugs.** CompDiff complements sanitizers' bug detection ability from three perspectives. First, for some kinds of bugs, CompDiff has a higher detection rate than the combined sanitizers. For example, on CWE-588 and CWE-758, CompDiff detects 99% and 93% of bugs while sanitizers in total only detect 49% and 36% of them. On CWE-457 and 665, although MSan in sanitizers specializes in detecting uninitialized variable uses, it only covers 7% of bugs while CompDiff identifies 92% of them. Second, even CompDiff fails to detect as many bugs as sanitizers in some classes of tests, it still discovers unique bugs that are missed by sanitizers. For example, for the memory errors shown in the first row, sanitizers detect 94% of bugs while CompDiff only achieves 63% detection rate. But there are 137 bugs that can only be covered by CompDiff. Third, CompDiff covers UBs that are not yet supported by sanitizers. None of the sanitizers discovers any bug on CWE-469, CompDiff, however, exposes them all.

➤ **Finding 3: CompDiff has the highest bug coverage compared to each individual sanitizer.** Compared to the results shown in columns "ASan", "UBSan", and "MSan", the second-to-last column shows that CompDiff can detect a diverse range of unstable code. On the contrary, each sanitizer only specializes in certain kinds of bugs. For example, MSan is only designed for uses of uninitialized variables and thus it cannot detect all other bugs. CompDiff's design, however, stems from the general consequence of UBs and thus has a higher overall bug coverage in principle.

➤ **Finding 4: CompDiff misses certain kinds of bugs.** Since sanitizers are designed for specific classes of bugs, they work better than CompDiff on them. For instance, UBSan reaches higher coverage than CompDiff in bugs related to integer errors and divide-by-zero. Because CompDiff only concerns a program's final output, an erroneous state resulting from these errors may not propagate to the output. This weakness is expected from the design choice of CompDiff. As we have been emphasizing, CompDiff is not to replace sanitizers but to complement them to cover more bugs.

➤ **Finding 5: CompDiff has no false positive.** To measure whether or not CompDiff reports any false positive, we also run CompDiff on *good* variant of each test. The result shows that CompDiff has no false positive, which is also the reason why we do not use a separate table for the result. For programs with deterministic output and correct compiler implementations, it is expected that outputs from the same input are identical across different compilations.

Summary. The above findings suggest that CompDiff is effective in detecting bugs related to UBs and has the highest overall bug coverage. Our results also confirm the strong discovery rate of each sanitizer on certain kinds of bugs. Compared to the combined sanitizers, on some of the UBs, CompDiff cannot detect as many bugs as them but still discovers additional unique bugs. We position CompDiff as a complementary tool to sanitizers. One should use both techniques to maximize the bug detection rate in practice.

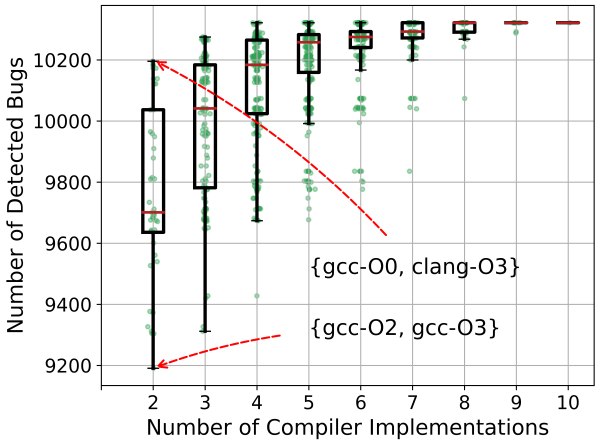


Figure 2.1: Number of bugs could be detected by each subset of compiler implementations.

2.3.2 *Impact of Reducing #Compiler Implementations*

By default, there are ten compiler implementations used in CompDiff, *i.e.*, `gcc` and `clang` with their respective optimization levels `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. In order to understand if it is necessary to use all of them in practice, we evaluate the number of bugs that can be detected by each subset of compiler implementations. We use the same Juliet tests as the previous evaluation. For each subset, *e.g.*, {`gcc-O0`, `gcc-O1`, `clang-O2`}, we modify CompDiff's configuration so that it only checks outputs from compiler implementations in the subset. We enumerate all possible subsets with sizes ranging from 2 to 10, and each subset does not contain duplicate compiler implementations.

The results are shown in Figure 2.1. We organize subsets according to their sizes. The X-axis means the size of each subset, *i.e.*, the number of compiler implementations in the subset. The Y-axis shows the number of bugs detected by each subset. We can find that with the increase of #compiler implementations, more bugs can be detected overall. For subsets of the same size, their detection ability varies a lot. For instance, when the number of compiler implementations equals 2, we have 45 subsets in combination. The number of bugs detected by them has a great difference, as shown in the first box in Figure 2.1. As annotated in the figure, the best performing subset with size=2 is {`gcc-O0`, `clang-O3`}. Intuitively, these two compiler implementations maximize compilation differences: both from different compilers, one is an unoptimizing compiler while the other is an aggressively optimizing compiler. The worst performing subset with size=2 is {`gcc-O2`, `gcc-O3`}. The reason is that they have the same basic compiler, and their optimizations are relatively similar. CompDiff's default setting achieves the best performance. Some small subsets, *e.g.*, 9, 8, or even 5, could detect nearly the same number of bugs as the default full size. We argue that this is due to the limited types of bugs in the Juliet tests. Discarding any compiler implementation risks missing bugs in practice.

Although the detection capability of small subsets may not be as good as the full size, they have lower run-time costs. For instance, using {`gcc-O0`, `clang-O3`} can detect ∼98% bugs compared to the full size, however, only has ∼20% run-time costs. If there are resource or time constraints, our results suggest that one can equip CompDiff with a smaller subset of compilers but should at least include two instances using different compilers and unoptimizing/(aggressively) optimizing optimizations.

2.3.3 CompDiff-*AFL++*

In this part, we evaluate the bug detection capability of CompDiff-AFL++ in real-world software.

Target projects. Table 2.4 lists details of all selected target projects. All these targets are well-studied and frequently used in the fuzzing community. They cover a broad range of functionalities, including network packet analyzers, binary file analyzers, multimedia file processing, programming language implementations, compression algorithms, *etc.* The sizes of these projects range from 10KLoC to 4.6MLoC, further emphasizing their diversity.

Experimental setting. We used the same experimental environment as previous experiments. To comprehensively evaluate CompDiff-AFL++'s capability, we equipped CompDiff with all ten compiler implementations. The initial seeds for targets are from their official test suites. Seeds of type images and videos are expanded with Mozilla Fuzzdata [129]. After a bug was found, we reported it to developers and relied on their feedback to triage all reports. As has been discussed, automated triage is challenging in the context of differential testing; we will discuss it further in Section 2.4. To compare with sanitizers, we compiled each program with ASan/UBSan and MSan to obtain sanitizer-enabled binaries. We then ran AFL++ on each

Table 2.4: Details of selected target projects.

Target	Input type	Version	Size(LoC)
tcpdump	Network packet	4.99.1	99K
wireshark	Network packet	3.4.5	4.6M
objdump	Binary file	2.36.1	74K
readelf	Binary file	2.36.1	72K
nm-new	Binary file	2.36.1	55K
sysdump	Binary file	2.36.1	10K
openssl	Binary file	3.0.0	702K
ClamAV	Binary file	0.103.3	239K
libsndfile	Audio	1.0.31	66K
libzip	Compress tool	v1.8.0	29K
brotli	Compress tool	v1.0.9	55K
php	PHP	7.4.26	1.4M
MuJS	JavaScript	1.1.3	18K
pdftotext	PDF	4.03	130K
pdftoppm	PDF	21.11.0	203K
jq	json	1.6	46K
exiv2	Exiv2 image	0.27.5	384K
libtiff	Tiff image	4.3.0	37K
ImageMagick	Image	7.1.0-23	655K
grok	JPEG 2000	9.7.0	127K
libxml2	XML	2.9.12	458K
curl	URL	7.80.0	13K
gpac	Video	2.0.0	597K

Table 2.5: Bugs detected by CompDiff-AFL++ on 23 open-source C/C++ projects.

	Unstable code due to undefined behavior					LINE	Misc.	Total
	EvalOrder	UninitMem	IntError	MemError	PointerCmp			
Reported	2	27	8	13	1	6	21	78
Confirmed	2	19	8	13	1	5	17	65
Fixed	2	15	6	12	1	5	9	52

binary and collected crashes found by sanitizers. All fuzzers had 24 hours timeout threshold and we repeated each fuzzing campaign 10 times.

Summary. Table 2.5 summarizes the number of bugs detected by CompDiff-AFL++. We categorize bugs based on their root causes. In total, CompDiff-AFL++ reported 78 bugs, 65 of which were confirmed by developers, and 52 were already fixed. The results demonstrate that CompDiff-AFL++ is effective and useful for identifying unstable code. Interestingly, as shown in the "LINE" and "Misc." columns, CompDiff-AFL++ detected not only undefined behaviors but also other real bugs due to various issues. We next analyze these bugs in detail. We guide our analysis with six consecutive research questions.

➤ RQ1: What kinds of unstable code does CompDiff-AFL++ find?

As shown in the "Unstable code due to undefined behavior" column in Table 2.5, we classify the found unstable code by their root causes into five categories. Note that unstable code that can be covered CompDiff-AFL++ is, in principle, not limited to these categories. For example, we did not find any bug related to CWE-469, on which CompDiff has shown its strong detection ability in Table 2.3. We anticipate CompDiff-AFL++ can detect a broader range of unstable code when evaluating more software. Next, we will show our analysis of each category.

EvalOrder. When the evaluation order of subexpressions has conflict side effects, the result becomes unstable. An example has been shown in Listing 3 in Section 2.1. These two bugs are all found in Tcpdump. After we reported these issues, the tcpdump developers quickly fixed it. They also manually diagnosed that the other 7 locations potentially had the same issue and fixed them as well. The developer commented that their fixes are just for *"less disruptive"* to the current code base and *"We should consider the cleaner long-term mechanism for 5.0 or later to get rid of static buffers"*.

UnitMem. Bugs due to uninitialized memories are classified into this category. We have shown an example in Listing 4 in Section 2.1. UnitMem bugs appear the most. The reason is that values of uninitialized variables depend on run-time memory layout, which often changes per binary. Some bugs in this category could be detected by MSan as well. We will compare our CompDiff with MSan on these bugs in **RQ3**.

IntError. Integer overflow/underflow can cause unstable code. The example in Listing 1 shows a case where a code fragment is discarded due to integer overflow. Sometimes, integer overflow may lead to inconsistent results across compiler implementations. For instance, the following code

```
1 int a, b;
2 long x, y;
3 ...
4 x = y + a * b;
```

contains a potential signed integer overflow in line 4 when `a*b` exceeds the range of `int`. In most cases, `a*b` is first calculated, and the result is then represented and stored to an `int`. However, some compiler implementations such as `clang-O1` first cast both `a` and `b` into `long` and then do the calculation. These two routines store different results to `x` when `a*b` overflows and thus becomes unstable.

MemError. Bugs in this category are caused by memory-related errors such as buffer-overflow and use after free. On the one hand, compilers can assume these errors never occur and transform code having MemError arbitrarily at compile time. On the other hand, these errors cause corrupted memory states and thus unstable program states. These bugs can be detected by ASan in principle. We will compare our CompDiff with ASan on these bugs in **RQ3**.

PointerCmp. Comparing pointers pointing to different objects / unions / structs is undefined. The comparison result can be either true or false, depending on the compiler's choice. CompDiff-AFL++ detected one such bug in readelf. We have discussed this bug in Listing 2 in Section 2.1.

LINE. The C17 standard specifies multiple permissible behaviors for the macro __LINE__ [14] (§6.10.4). Interestingly, the interpretation of __LINE__ is implementation-defined behavior, meaning that different compiler implementations may have divergent interpretation results. CompDiff-AFL++ detected such inconsistencies in programs including `readelf`, `ImageMagick`, `Wireshark`, `libtiff`, and `php`. For instance, for the following PHP code with the bug in line 3,

```
1 <?php
2 $a = 0;
3 var_dump($b::class);
4 ?>
```

some php interpreters compiled from different compilers incorrectly label line 2 instead of 3 as buggy.

Miscellaneous. Surprisingly, CompDiff-AFL++ detected 21 real bugs that are not due to undefined behavior. Their root causes are miscellaneous. One interesting category is compiler bugs/issues, where compilers instead of application programs are blamed for the divergent output. We will further discuss this kind of bug in the next RQ. CompDiff detected many other program-specific issues including, but not limited to, bad random value (libtiff), printing pointer address instead of value (objdump), and even unknown reasons (wireshark). All of these issues lead to divergent outputs on the same input, and thus none of them are false positives. These bugs demonstrate CompDiff's strong ability in exposing unstable issues.

➤ RQ2: Did CompDiff-AFL++ detect any compiler bug or issue?

In our CompDiff's design, we assume compilers are bug-free. In practice, compiler bugs rarely affect real-world programs' integrity [101]. According to our unstable code definition, compiler bugs or issues can indeed cause divergent program outputs and thus be caught by CompDiff. On the 23 real-world programs, we found 3 compiler bugs due to miscompilation and 4 compiler issues due to floating point imprecision. Next, we discuss them in detail.

Compiler miscompilation. CompDiff reported two compiler miscompilations in `gcc` and one in `clang`, all of which were found during fuzzing `MuJS`. Unlike other bugs introduced by program developers, compilers are blamed for these bugs. After we reported these bugs, the `MuJS` quickly confirmed these issues and proposed solutions to avoid compiler miscompilations in their code temporarily. Although compiler issues have been detected, we do not anticipate CompDiff can be used to find compiler miscompilations intensively. Nevertheless, these issues are real bugs that affect program correctness. CompDiff helps identify them and guarantees the program's integrity across compilations.

Floating-point imprecision. Different compilers or optimizations may use different strategies to calculate floating point data. For example, `clang-O3` sometimes transforms `pow()` to the more efficient `exp2()` libcall and has different decimal results from others. Rounding strategies in different compiler implementations may also cause divergent results. Strictly speaking, this is not a program bug but a compiler issue. We reported four such cases, and developers confirmed three of them. Only `brotli`'s developers committed to fixing the reported bug, which was because floating-point imprecision affected the internal state of a compression algorithm, making the compressed file different across compiler implementations. We do not consider any of these bugs to be false positives because they indeed lead to inconsistent results across compilations. Compared to other bugs, floating-point imprecision is relatively less serious in most cases.

➤ RQ3: How many bugs found by CompDiff-AFL++ can be covered by sanitizers?

Some of the detected bugs can, in principle, be captured by sanitizers. Specifically, *MemError* by ASan, *IntError* by UBSan, and *UninitMem* by MSan. For each bug detected by CompDiff, we check if it can be discovered by sanitizers. Recall that AFL++ with ASan, UBSan, and MSan were run on each target 10 times, and we collected all the reports they produced. For each bug reported by CompDiff, we performed manual analysis to identify whether or not sanitizers' reports cover it. Table 2.6 shows the summarized result. Out of 78 bugs detected by CompDiff, 42 of them can also be covered by sanitizers. The left 36, however, cannot. Specifically, for *MemError* and *IntError*, ASan and UBSan successfully detected all of them. MSan, however, only exposed 21 out of 27 *UninitMem* bugs. In fact, we also observed that many bugs detected by sanitizers during fuzzing cannot be detected by CompDiff, either. The observation is aligned with our findings in the Juliet tests. As we have been emphasizing, CompDiff is not to replace sanitizers but to complement them in exposing more bugs in practice. The unique 36 bugs detected by CompDiff support our claim.

➤ RQ4: Impact of #compiler implementations in CompDiff-AFL++.

Table 2.6: Of all the bugs detected by CompDiff, the number of bugs that can also be discovered by sanitizers.

	Sanitizers				CompDiff
	ASan	UBSan	MSan	**Total**	
MemError	13	-	-	13	13
IntError	-	8	-	8	8
UninitMem	-	-	21	21	27
Remaining bugs	-	-	-	0	30
Total				42	78

We have analyzed on the Juliet tests the impact of different subsets of compiler implementations in CompDiff. To examine if a consistent tendency can be observed in CompDiff-AFL++, we also evaluate each subset of compiler implementations on the 78 real bugs. Figure 2.2 presents the results. Compared to our previous results in Figure 2.1, similar conclusions can be drawn: More compiler implementations bring higher bug detection rates in CompDiff-AFL++; different compilers with unoptimizing and aggressively optimizing levels can bring us the best performance while similar compiler implementations are less effective.

➤ **RQ5: Handling non-deterministic or multi-threaded programs in CompDiff-AFL++.**

Recall that CompDiff-AFL++ is designed to find unstable code in programs with deterministic output. Non-deterministic programs are typically concurrent or multi-threaded programs. They can have either deterministic outputs, *i.e.*, running the program on the same input always emits the same output, or inconsistent outputs, *i.e.*, outputs may change per run. For non-deterministic or multi-threaded programs, CompDiff can handle them as long as they have deterministic output. In fact, among our evaluated 23 real-world programs, 6 of them are non-deterministic or multi-threaded programs, *i.e.*, `tcpdump`, `wireshark`, `MuJS`, `ImageMagick`, `grok`, and `gpac`. In total,

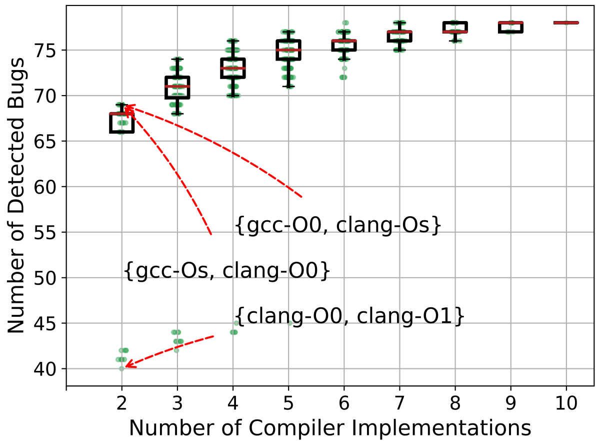


Figure 2.2: Number of bugs could be detected by each subset of compiler implementations by CompDiff-AFL++.

CompDiff-AFL++ found 23 bugs in them, which shows that CompDiff is capable of handling non-deterministic programs that have deterministic outputs.

For non-deterministic programs without deterministic output, our experience is that many of the non-determinism happen when programs deliberately include random numbers or timestamps in their outputs. Such non-determinism can be easily eliminated with a post-processing script on the program's output. For example, the output of `wireshark` includes the timestamp when it generates warnings:

```
10:44:23.405830 [Epan WARNING]
```

Different binaries are thus emitting inconsistent warning messages. To filter out these values, we used a regular expression to match and remove all these timestamps in its outputs.

➤ **RQ6: False positives of CompDiff-AFL++.**

Theoretically, CompDiff has no false positives since all test inputs saved by CompDiff are guaranteed to trigger discrepancies across at least two compiler implementations. However, we still need to deal with cases where some but not all binaries timeout. For efficiency reasons, AFL++ deliberately terminates a binary after a timeout threshold. Such terminations will inevitably truncate a binary's output, causing output discrepancy between terminated binary and timed-out binary. In our CompDiff-AFL++, when a generated input times out on partial binaries, we let CompDiff-AFL++ save it first and then increase its timeout threshold until it terminates. It is theoretically possible that a binary hangs forever. However, our experience is that as long as one of the compiled binaries terminates, others will terminate eventually but may have a longer execution time.

As we discussed before, false positives are also possible when using CompDiff on non-deterministic programs that do not have deterministic output. We have shown in RQ5 that many of the non-determinism in the output have fixed patterns such as random number and timestamp and can thus be eliminated. We consider the general handling of programs with non-deterministic output as a limitation of CompDiff and discuss further in Section 2.4.

2.4 DISCUSSION

Fault localization and bug report. When a program failure is found, it is beneficial to automatically and accurately localize the root cause in the source code. Fault localization techniques, in general, although extensively studied [154], are not yet practical [11, 117]. Sanitizers provide function stack traces to help developers pinpoint the root causes of crashes. Since bugs found by CompDiff do not necessarily lead to crashes, such stack trace-based approaches are not applicable. As all tests reported in CompDiff result in different outputs across binaries, it is possible to compare execution traces from different binaries to pinpoint the root cause. Aligning executions on two binaries is challenging in general. Although CompDiff has the advantage that all binaries are compiled from the same source code, compilers, especially optimizations, will result in huge differences in control flows and variable values. It would be interesting for future work to explore how to accurately and efficiently align and compare multiple execution traces.

Although we do not analyze the root cause of bugs with CompDiff, our current bug reports are useful for developers to diagnose and fix bugs. In our bug reports, we include the following information: 1) *test input* that triggers the bug, 2) *two or more compiler configurations* that can be used to reproduce the bug, and 3) *the divergent outputs* on the provided test input. With our current reports, at the time of writing, 52 out of the 78 reported bugs had already been fixed by developers. Thus, there is clear evidence that our reports are useful as developers are able to use them to diagnose these issues quickly.

Limitations. CompDiff has detected many unknown bugs in real-world software. All these bugs lead to inconsistent/incorrect outputs. CompDiff incurs no false positive in programs with deterministic output. Although all UBs, by definition, could lead to unstable code, existing compiler implementations may not exploit all of them. Thus, CompDiff cannot detect all UBs in practice. For unstable code, CompDiff cannot discover all of them for two reasons. First, not all erroneous states of unstable code propagate to final outputs. Since CompDiff concerns output only, there is no way to detect vanishing errors. Second, not all unstable code leads to inconsistent behaviors. At run-time, binaries may enter the same erroneous state, *e.g.*, choosing the same uninitialized value, and emitting identical but incorrect output. Extending CompDiff to a program's internal states would incur huge analysis overhead and might be generally infeasible and impractical. The succinct design of CompDiff guarantees its generality and scalability. Another limitation of CompDiff is in handling non-deterministic programs that have non-deterministic output. An example is a multi-threaded program where multiple threads concurrently print into the standard output while there is no requirement on the order. In such cases, divergent outputs across runs are normal, and thus CompDiff will not be useful in detecting errors. However, during our testing, we did not encounter such cases, and all our targeted non-deterministic programs have deterministic output.

Overhead. Our experimental results suggest that we should enable all ten compiler implementations when using CompDiff in practice. The run-time overhead is roughly 10x normal execution. However, our evaluations on the Juliet tests and real-world software also reveal the fact that a smaller subset of compiler implementations can give us a similar bug detection rate. For example, using only `clang-O0` and `gcc-Os` in CompDiff-AFL++ can discover 69 out of total 78 bugs. The run-time overhead can be reduced from roughly 10x to 2x normal execution. When using CompDiff in practice, we suggest users enable as many compiler implementations as possible to maximize its bug-finding capability. If resources are constrained, users should enable at least different compilers with diverse optimization levels.

Improvements and future work. The current design of CompDiff-AFL++ keeps the fuzzer's core logic and applies differential testing on each generated input. Its effectiveness largely depends on the quality of test inputs.

AFL++ utilizes code coverage feedback from executions to guide its mutation strategies. If we incorporate the divergence observed from different binaries into the feedback, AFL++ could potentially generate more unstable code-triggering test inputs. NEZHA [119] takes a similar approach by exploiting behavioral asymmetries between multiple programs to improve AFL's efficacy in generating inputs that are likely to trigger semantic bugs. Its design is for different implementations on the same specification such as OpenSSL and LibreSSL. In CompDiff, binaries are compiled from the same source code. Aligning execution paths and finding execution divergence is much easier than NEZHA's situation. Integrating execution divergence into fuzzers' feedback may enable CompDiff to find a lot more unstable code. We believe this will be an interesting future work to enhance compiler-driven differential testing further.

2.5 RELATED WORK

Differential testing. Researchers have leveraged differential testing to find semantic bugs across many types of programs, such as database management systems [126], Java Virtual Machine (JVM) implementations [23], REST APIs [47], and compilers [66]. Incorporating different testing into fuzzing engines is also gaining more and more interest in discovering functional bugs. For example, HeteroFuzz [164] detects platform-dependent divergence for heterogeneous applications running on both CPU and FPGA. DifFuzz [108] discovers side-channel leakages by analyzing two executions on the same binary. DIFUZZRTL [59] finds CPU bugs by differentially comparing multiple CPU RTLs. It develops a novel register coverage for higher fuzzing efficiency and efficacy.

Each of these differential fuzzing efforts focuses on one specific domain. In contrast, CompDiff utilizes a generally applicable test oracle that concerns a totally different kind of bugs, *i.e.*, unstable code. Another notable difference is that most of them incorporate new designs for fuzzers in order to improve the possibility of discovering bugs. CompDiff does not make changes to fuzzers core logic and has been proven to be effective in finding interesting bugs. Designing new feedback for CompDiff should be an interesting research direction for future exploration.

N-version programming. The concept of N-version programming (NVP) was first introduced in 1978 by Chen and Avizienis [18]. It aims to improve software fault tolerance by having N independent individuals or groups implement the same specification. Recent and practical applications of NVP include opportunistically leveraging existing diverse software implementations. For example, FROST [142] executes multiple replicas with complementary thread schedules to protect a program from data race errors; VARAN [56] utilizes system call level synchronisation to realize N-version execution systems; many differential testing approaches discussed earlier are also instances of NVP, such as Chen *et al.* [23] that leverages different JVM implementations like Oracle's HotSpot and IBM's J9. COMPDIFF employs different compiler implementations to obtain N-version binaries of a program. We do not require any modification to a program's source code or its execution environment. In contrast to many differential testing approaches, where additional implementations of a target program are necessary, COMPDIFF, however, is not subject to this requirement. We argue that our employment of compiler implementations to get multiple replicas is novel and unique compared with the existing NVP schemes.

Finding undefined behavior. Undefined behavior in C/C++ covers a wide range of illegal program states. Apart from sanitizers, there are also many other popular tools such as Dr. Memory [13] and Valgrind [107] that require no compile-time instrumentation and detect errors at the binary level. Static tools such as STACK [152], Infer [104], and Cppcheck [25] cal also cover frequently occurring UBs. COMPDIFF is orthogonal to them as 1) as a dynamic tool, COMPDIFF discovers bugs that are beyond the reach of static tools, and 2) developers can always use static tools, as well as dynamic tools, in different development stages for finding more bugs.

3 ACCELERATING FUZZING THROUGH PREFIX-GUIDED EXECUTION

As the evaluation in Chapter 2 has shown, an effective undefined behavior detection requires not only the underlying test oracle like CompDiff but also a fuzzer to generate and execute numerous test inputs. The most widely used and effective general-purpose fuzzer is coverage-guided fuzzer (CGF) [1, 96], such as AFL [161] and its advanced version AFL++[35]. CGF executes all mutated tests from seed inputs to expose coverage-increasing tests. However, executing all mutated tests incurs significant performance penalties—most of the mutated tests are discarded because they do not increase code coverage [105]. Thus, determining if a test increases code coverage *without actually executing it* is beneficial but a paradoxical challenge. In this chapter, we introduce the notion of *prefix-guided execution* (PGE) to tackle this challenge.

The work in this chapter was published in [76].

Key idea. PGE leverages two key observations: (1) Only a tiny fraction of the mutated tests can increase coverage, thus requiring full execution, and (2) whether a test increases coverage may be accurately inferred from its partial execution. PGE monitors the execution of a test and applies early termination when the execution prefix indicates that the test is unlikely to increase coverage. At a high level, for a CGF-style fuzzer, PGE works as a replacement for the fuzzer's execution engine. Instead of fully executing all tests, PGE selects prefix-interesting ones to execute them fully and discards all the others. A proper prefix length is crucial for the performance of PGE. However, for a general target, it is challenging to determine a prefix length that can identify coverage-increasing tests while being as small as possible. Utilizing static/symbolic analysis to reason about constraints between different program locations and decide a proper prefix length, although plausible in theory, is difficult to (1) scale to complex real-world code (which is why fuzzing is much more widely adopted) and (2) handle low-level code (*e.g.*, binaries) for settings where source code is unavailable. To tackle this challenge, we propose a sampling-based search algorithm to infer prefix lengths with low overhead dynamically. Our algorithm simulates the current fuzzing loop and finds the prefix length that can recall a desired amount of interesting tests.

As Section 3.1 will show, a relatively short execution prefix can help identify a large proportion of coverage-increasing tests. Since finding bugs is the ultimate goal of fuzzers, we will also show that execution prefixes can also effectively identify bug-triggering tests.

Main contributions. Our contributions are summarized as follows:

- We study the correlation between execution prefixes and the coverage increasingness of tests by quantifying their strong connection across nine real-world projects.
- We propose the novel, general technique of prefix-guided execution (PGE) to speed up and improve fuzzing by early terminating executions that are unlikely to increase coverage.
- We present a simple prefix length search algorithm for finding a proper prefix length to guide PGE effectively.
- We realize PGE in AFL++-PGE, a prototype on top of AFL++, and extensively evaluates AFL++-PGE to demonstrate its utility in terms of fuzzing cost reduction, coverage increasing, and bug detection.

The artifact for PGE, including all source code and data, is permanently available [74].

3.1 OBSERVATIONS ON CGF

This section introduces several key observations on coverage-guided fuzzing (CGF) that PGE builds upon. Without loss of generality, given a target binary and an initial seed pool, the high-level workflow of a fuzzer is as follows:

(1) **Seed selection.** Select one seed from the seed pool according to predefined strategies.

(2) **Test generation.** Mutate the seed to generate a large number of tests.

(3) **Execution and feedback collection.** Execute each of the tests on the target binary and collect coverage feedback, queue a test to the seed pool if it increases coverage, and report it when it causes an execution failure, *e.g.*, crash.

Table 3.1: Rates of Coverage-increasing Tests Out of All Tests Generated in One Hour By AFL++.

libpng	*libsndfile*	*libtiff*	*libxml2*	*lua*	*openssl*	*php*	*poppler*	*sqlite3*	**avg.**
0.03%	0.05%	0.11%	0.21%	0.24%	1.00%	0.25%	2.12%	0.11%	**0.46%**

(4) **Go to step (1) and repeat.**

The workflow shows that coverage-guided fuzzers work in a loop. In each loop, the selected seed will be mutated and blindly executed a tremendous amount of times to filter out those coverage-increasing ones. However, as has been reported by previous work [105], the overwhelming majority of test cases are non-coverage-increasing. Although executing the target binary on them wastes most of the allocated resources, fuzzers still have to do so as it is, by now, the only way to understand whether a test increases coverage.

However, as we will show in this section, it is possible to infer coverage-increasing property without full test execution accurately — partial execution suffices in most cases. We use 21 programs from nine real-world projects in Magma benchmark [52] to demonstrate the key observations that our design relies on. We averaged the results of programs from the same project. We choose, as our target fuzzer, the most popular coverage-guided fuzzer AFL++ [34] on which many fuzzers are built. All experiments are run with the same setup as that for our later performance evaluation.

Observation 1: Only a tiny fraction of tests are coverage-increasing.

We run AFL++ on each program for an hour. All experiments are repeated 12 times with different initial seeds. We log the total number of generated tests during fuzzing. The number of coverage-increasing tests is learned by counting the new seeds that AFL++ appends into the seed pool. In AFL++, a test is interesting when it reaches new edges or significantly increases edge counts. We treat both cases as coverage-increasing. Table 3.1 shows the averaged rates of coverage-increasing tests during one hour of fuzzing. On seven out of nine projects, less than 1% of tests generated by AFL++

increased coverage. On average, AFL++ has 0.46% coverage-increasing tests out of all tests in each one-hour trial. That means during the first hour of fuzzing, averagely only 4 out of 1000 tests are actually interesting and worth executing. If we are able to choose only those interesting ones, the execution cost of the remaining large number of tests can be saved.

Observation 2: Execution prefixes correlate highly with a test's coverage increasingness.

Mainstream coverage-guided fuzzers use basic block edges as their coverage metrics (26 out of 27 as reported in [106]), so our study focuses on edge coverage as well. For a coverage-guided fuzzer, we denote an execution trace T^k as an ordered temporal sequence $T^k = \langle e^k_0, e^k_1, \ldots, e^k_{n-1} \rangle$, where $e^k_i, i \in [0, n)$ is the i-th edge this execution accessed. Tracing all temporal execution traces during fuzzing is extremely costly. Instead, AFL++ uses a global coverage bitmap with counts to store the accessed edges such that an execution is identified by $E^k = multiset(T^k) = \{e^k_0, e^k_1, \ldots, e^k_{n-1}\}$[2], where the item ordering is ignored and duplicated edges are included. Before introducing our observation, we first define the notion of *execution prefix* used in this work.

Definition 2 (Execution Prefix). *Given an execution trace T^k of length n and predefined prefix length $l \leq n$, the* execution prefix *of T^k is defined as $\Pi^k(l) = multiset(\langle e^k_0, e^k_1, \ldots, e^k_{l-1} \rangle) = \{e^k_0, e^k_1, \ldots, e^k_{l-1}\}$.*

This definition shows that an execution prefix $\Pi^k(l)$ is a subset of the corresponding full execution E^k. This feature allows us to reuse the bitmap structure in AFL++ and obtain $\Pi^k(l)$ by terminating the execution when accessing the l-th edge (See technical details in Section 3.2.2). AFL++ considers a test *interesting* when it increases coverage. Since our goal here is to discover the connection between execution prefixes and such interestingness of full executions, we define next the notion of *interesting execution prefix*.

Definition 3 (Interesting Execution Prefix). *Given a predefined prefix length l, m seen executions $\mathcal{E} = \{E^0, E^1, \ldots, E^{m-1}\}$, and their execution prefixes $\mathcal{P} = \{\Pi^0(l), \Pi^1(l), \ldots, \Pi^{m-1}(l)\}$. The subsequent execution prefix $\Pi^m(l)$ of E^m is interesting iff $\Pi^m(l) \notin \mathcal{P}$.*

2 A *multiset* contains duplicated items while item ordering is ignored (https://en.wikipedia.org/wiki/Multiset).

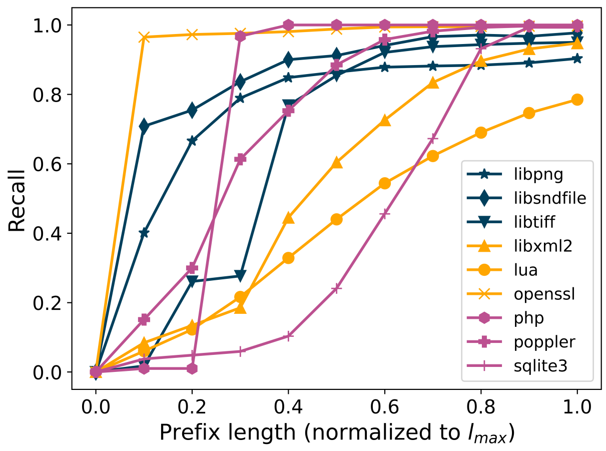


Figure 3.1: Recall of coverage-increasing executions from interesting execution prefixes.

This definition shows that when determining whether an execution prefix $\Pi^m(l)$ is interesting or not, each seen execution prefix in $\mathcal{P}$ is taken into account. In our implementation, we cached the hash values of all execution prefixes for such lookups (Technical details in Section 3.2.3). Suppose that an execution E^m is interesting *w.r.t.* code coverage implies its prefix $\Pi^m(l)$ is also interesting *w.r.t.* Definition 3. We could then use execution prefixes to filter out all uninteresting tests without fully executing them. Since it takes less time to obtain execution prefixes than full executions, this strategy can potentially boost fuzzers' efficiency.

We now empirically validate this assumption to understand how commonly it holds in practice. Execution prefixes can be obtained with a modified AFL++, which terminates an execution when it reaches a predefined prefix length limit. Technical details will be shown in Section 3.2. Experiments on each program are done 12 rounds with different initial seeds. For each round of experiment on a program, we choose one seed and follow the steps below:

1. Run AFL++ on one seed with a one-hour timeout and collect all generated tests. Identify the length of each execution and label all coverage-increasing executions.

2. Set the max prefix length l_{max} as the average length of the collected executions. In principle, as long as setting $1.0 \cdot l_{max}$ as the prefix length

can retrieve nearly all interesting tests, the selected metric for l_{max} is eligible. Our preliminary experiments show that setting both mean and median prefix lengths as l_{max} is qualified. In our implementation, we choose to use the mean value because, unlike the median, it does not need to track all intermediate values.

3. Select 10 evenly distributed prefix lengths $l \in \{0.1 \cdot l_{max}, 0.2 \cdot l_{max}, \dots, 1.0 \cdot l_{max}\}$. For each prefix length, rerun AFL++ on the same set of tests to collect their execution prefixes. Identify all interesting execution prefixes.

4. Out of all coverage-increasing executions, count how many of them also have interesting execution prefixes, *i.e.*, the recall of coverage-increasing executions by interesting prefixes.

Figure 3.1 shows the average recall of coverage-increasing executions by interesting execution prefixes. The top six curves demonstrate that for these six projects, recall rates grow exponentially with the increase of prefix length. Short prefixes are less effective on `lua` and `sqlite3`, but longer prefixes still do. This experimental result reveals that a short execution prefix suffices to locate most of the coverage-increasing tests. On six out of nine projects, achieving 70% recall on average needs prefixes $\leq \frac{4}{10}$ in length of the full executions, and $\leq \frac{6}{10}$ for `libxml2`. Suppose that the prefix length to achieve a high recall is known to a fuzzer. It can partially execute most tests while only fully executing those with interesting prefixes.

Observation 3: Not all bug-triggering tests are coverage-increasing.

For all tests mutated from one seed, suppose that an ideal fuzzer is able to identify coverage-increasing ones and only executes them. This fuzzer would achieve the same code coverage as executing them all. However, finding bugs is the ultimate goal of a fuzzer. We cannot guarantee that this ideal fuzzer can find the same number of bugs unless bug-triggering tests are also coverage-increasing.

To find out if bug-triggering tests could also increase coverage, we run AFL++ on the same set of programs for 48 hours, then collect all coverage-increasing tests (placed in "queue" directory) as well as unique crash-triggering tests (placed in "crashes" directory). We then rerun AFL++ on all these tests according to their creation timestamps to simulate the fuzzing

Table 3.2: The Number of Selected Tests Out of All Bug-triggering Tests by Different Metrics.

	libpng	libsndfile	libtiff	libxml2	lua	openssl	php	poppler	sqlite3
Total	62	32	252	921	6	18	34	212	68
Coverage	8	6	10	377	2	4	5	52	14
Pattern	62	32	250	900	4	18	34	211	66

process and check if a bug-triggering test increased code coverage at their creation time.

The "Total" row in Table 3.2 lists the average number of bug-triggering tests found in each project. Of those, the "Coverage" row shows the number of coverage-increasing tests. We can see that only a small fraction of bug-triggering tests increased code coverage. In this case, executing only coverage-increasing tests during fuzzing may miss many bugs. To address this issue, we need to relax the restrictions on interesting executions, which should contain not only coverage-increasing tests but bug-triggering tests as well. We choose to use the whole execution pattern, which can be viewed as an execution prefix with the maximum prefix length (*i.e.*, $l = n$ in Definition 2). Such patterns meet all of our requirements since (1) theoretically a coverage-increasing execution always implies its execution pattern being interesting, *i.e.*, never being seen; (2) empirically the "Pattern" row in Table 3.2 shows that most of the bug-triggering tests also have interesting patterns.

Recall Observation 2 discussed earlier. Since we relaxed the scope on interesting executions, we now need to validate if execution prefixes still have a strong connection with new execution patterns. We run the same experiments as in Observation 2 but change target executions from coverage-increasing to pattern-interesting. Figure 3.2 shows the results. Execution prefixes keep the correlation tendency across projects but have relatively lower recall values. This is inevitable due to our relaxations but does not affect the overall effectiveness.

As will be detailed in Section 3.2.4, the whole execution pattern for each execution will only be used in the prefix search procedure of PGE.

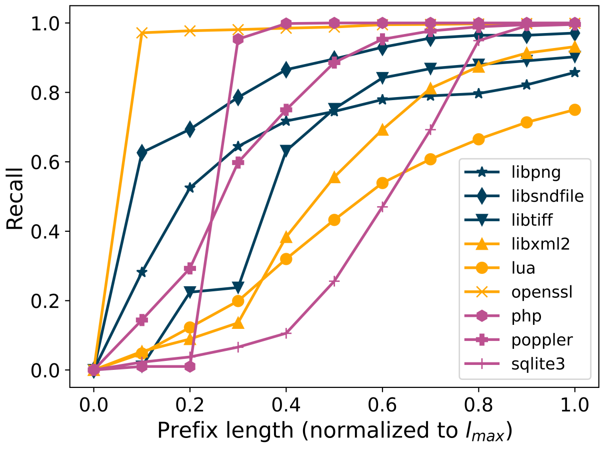


Figure 3.2: Recall of pattern-interesting executions from interesting execution prefixes.

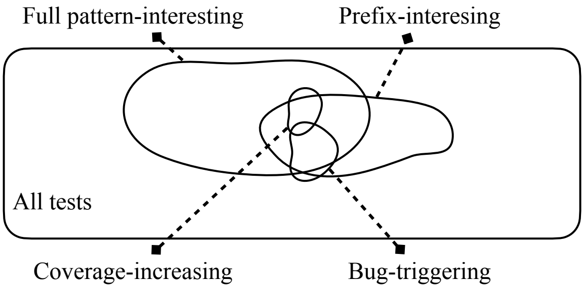


Figure 3.3: Illustrative relations from observations.

During prefix search, PGE evaluates each prefix by its recall capability of all pattern-interesting whole executions.

Summary: The above observations empirically show the relationships between coverage-increasing executions, bug-triggering tests, pattern-interesting full executions, and prefix-interesting executions. Figure 3.3 illustrates their relations pictorially. The target of our proposed PGE is to identify prefix-interesting tests and execute fully on them instead of all tests. Note that

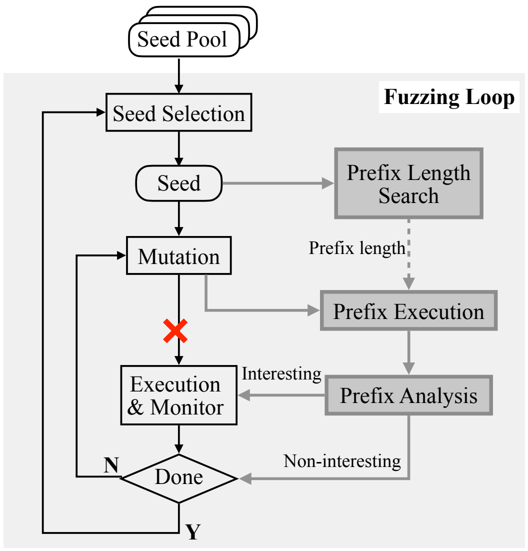


Figure 3.4: Overview of a coverage-guided fuzzer augmented with prefix-guided execution.

this figure is only for conceptual illustration. Size ratios in the figure do not reflect their real values.

3.2 APPROACH

This section introduces technical details for prefix-guided execution (PGE) and how we augment a coverage-guided fuzzer with PGE.

3.2.1 *Fuzzing with* PGE

Figure 3.4 shows the high-level workflow of a coverage-guided fuzzer augmented with PGE. The three grey blocks highlight the key extensions that we make to the standard coverage-guided grey-box fuzzing. This new workflow illustrates that the fuzzing framework has not been altered and the PGE extension is generally applicable.

Prefix Execution is used to obtain the execution prefix of a test. Given a test and prefix length l, this module executes the target binary and terminates it whenever the execution visits l edges. As long as l is smaller than the actual execution length, such early termination reduces the time required for execution. This is a logic module that exists in the form of extra instrumentation code in the target binaries. (See details in Section 3.2.2.)

Prefix Analysis is for analyzing whether or not a given execution prefix is interesting. Interesting prefixes refer to those that have never been observed since the start of the current fuzzing loop. Only tests showing interesting prefixes will be fully executed, while tests with non-interesting prefixes will be discarded (See details in Section 3.2.3).

Prefix Length Search estimates a proper prefix length for the current fuzzing loop. Recall the observations in Section 3.1. Different prefix lengths correspond to different recall rates of pattern-interesting executions. When validating the observations, the concrete relationships between prefix lengths and recalls are from the post-processing of fuzzing loops. Now, when employing them in fuzzing, we need to learn such relationships before the start of a fuzzing process. We propose a sampling-based search algorithm to find an appropriate prefix length that can reach a given recall rate. (See details in Section 3.2.4).

Algorithm 2: Fuzzing with PGE

```
Input: Seeds $\mathcal{S}$, Recall rate $r$ .
1  while $\neg$Abort() do
2      $s \leftarrow$ SelectSeed($\mathcal{S}$)
3      $l \leftarrow$ PrefixLengthSearch($s,r$) // Search for prefix length
4      $p \leftarrow$ AssignEnergy($s$)
5      for $i$ from 0 to $p$ do
6          $s' \leftarrow$ Mutate($s$)
7          $\Pi^{s'}(l) \leftarrow$ PrefixExecution($s',l$) // Get execution prefix
8          if PrefixAnalysis($\Pi^{s'}(l)$) == interesting then
9              $E^{s'} \leftarrow$ FullExecution($s'$) // If new execution prefix
10             if IsInteresting($E^{s'}$) then
11                 add $s'$ to $\mathcal{S}$
12         else
13             continue
```

Algorithm 2 shows an algorithmic sketch of how fuzzing with PGE works. The grey boxes highlight our extensions. The fuzzer is provided with a set of seeds $\mathcal{S}$ and a target recall rate r. During each fuzzing loop (the top *while* loop), a seed s is selected from $\mathcal{S}$ (line 2). All tests generated within the present fuzzing loop are derived from this seed. The fuzzer then searches for the smallest possible prefix length l that may reach the recall rate r, which is implemented in `PrefixLengthSearch` (line 3). An energy p is assigned to the seed, which represents the number of tests that will be generated according to the defined mutation operators (lines 4-6). For each new test s', the fuzzer partially executes the target binary, collects execution prefix $\Pi^{s'}(l)$ of length l (line 7), and goes to different branches:

1. If $\Pi^{s'}(l)$ is interesting, *i.e.*, having never been seen in the current fuzzing loop, the fuzzer invokes the normal execution and monitor procedures to, *e.g.*, add s' to $\mathcal{S}$ if it increases code coverage, or cache s' if it results in a unique crash/hang. (lines 8-11)

2. Otherwise, the fuzzer continues with the next test. (line 13)

According to the observations from Section 3.1, only a tiny fraction of tests explored in a fuzzing loop are interesting, *i.e.*, coverage-increasing or bug-triggering. If there were an ideal procedure that could select interesting tests without any execution overhead, a fuzzer, when equipped with such a procedure, could be dramatically more efficient. Such an ideal procedure certainly does not exist. `PrefixExecution` and `PrefixAnalysis` are designed to obtain a practical instance of such a procedure with partial execution overhead. For each test with an interesting execution prefix, the execution overhead increases from `FullExecution` to "`PrefixExecution` + `FullExecution`"; while for each test with an uninteresting execution prefix, the execution overhead decreases from `FullExecution` to `PrefixExeuction`. The efficacy of prefix-guided execution critically depends on the proportion of interesting prefixes.

As will be shown in the evaluation, for most generated tests, AFL++ with PGE will execute the second branch above, *i.e.*, discarding the tests. A large proportion of uninteresting prefixes make the fuzzer discard most tests without fully executing them, thus improving fuzzing efficiency.

3.2.2 *Prefix Execution*

Given a prefix length l, `PrefixExecution` is able to terminate the target binary when it visits l edges. This is achieved by augmenting the standard

```
void
__sanitizer_cov_trace_pc_guard(uint32_t *guard) {
  __afl_area_ptr[*guard]++;
}
```

(a) Original AFL instrumentation code

```
void
__sanitizer_cov_trace_pc_guard(uint32_t *guard) {
  __afl_area_ptr[*guard]++;
  // Update the global prefix counter.
  __afl_prefix_cntr++;
  // Terminate the execution
  // when reaching the given prefix length.
  if (__afl_prefix_cntr == __afl_prefix_len)
    _exit(0);
}
```

(b) Extended AFL-PGE instrumentation code

Figure 3.5: Original and extended AFL++-PGE instrumentation codes for edge coverage.

coverage-tracing instrumentation. Next, we will use AFL++ as the target fuzzer to show how to support prefix execution in AFL++ instrumentation.

We use the "trace-pc-guard" mode in AFL++ for edge coverage tracing. This mode is back-ended by LLVM SanitizerCoverage [94], which can insert calls to user-defined functions at the level of basic block edges. Figures 3.5 (a) and (b) show, respectively, the user-defined functions used in AFL++ and AFL++-PGE. In AFL++, edge coverage is reported by calling the function `__sanitizer_cov_trace_pc_guard()` with a unique identifier `guard` to increment the corresponding entry in `__afl_area_ptr` coverage map.

To count the number of edges visited, the extended instrumentation code increments a global counter `__afl_prefix_cntr` which is initialized to 0 at the start of an execution. The target prefix length is passed to `__afl_prefix_len` via a shared memory between the fuzzer and the target binary. When `__afl_prefix_cntr` reaches the limit, *i.e.* `__afl_prefix_len`, the ongoing execution will be terminated. At this moment, the coverage map `__afl_area_ptr` stores the execution prefix.

Note that the extended AFL++-PGE instrumentation code also supports full execution by setting `__afl_prefix_len=-1`, making the termination code unreachable. So, `PrefixExecution` and `FullExecution` in Alg. 2, in fact, share the same instrumented target binary.

3.2.3 *Prefix Analysis*

We use a global hashmap `prefixMap` to record all prefixes that have been observed in the current fuzzing loop. For each new execution prefix, `PrefixAnalysis` first calculates its hash[3] value to index `prefixMap`: If the indexed entry is 1, it returns non-interesting; if the indexed entry is 0, it sets it to 1 and returns interesting. Note that, for every newly selected seed, `prefixMap` is emptied and rebuilt, which is due to the fact that the coverage map changes over time, and the proper prefix length for the same seed may change accordingly. Because prefix lengths are not identical for different seeds, it is less meaningful to share `prefixMap` among them.We also use another such hashmap `fullMap` to record all pattern-interesting full executions. Since interesting full executions can be shared across seeds, different from `prefixMap`, the `fullMap` is only initialized at the beginning of fuzzing.

3.2.4 *Prefix Length Search*

To benefit from prefix-guided execution, a proper prefix length *w.r.t.* target recall r should be learned before fuzzing starts. The recall r describes the capability of a prefix in selecting interesting tests. A prefix length with higher recall implies that it is likely to select more interesting tests. However, for a recall r, the ground-truth minimal prefix length can only be learned by fully executing all candidate tests, thus it has no practical value.

To practically estimate a proper prefix length, we propose a sampling-based search algorithm. Instead of running all tests, the algorithm samples a tiny fraction, say 5%, runs them both fully and partially, and learns the recall capabilities of different prefix lengths. Alg. 3 details the prefix length search algorithm. It consists of three main stages:

1. Same as the common fuzzing procedure, energy is assigned to the given seed but is reduced by a factor sr (line 1). Tests from p times mutations are then cached (lines 2-4).

3 We use the MurmurHash3 [3] hash function supported by AFL++.

Algorithm 3: Prefix Length Search

Global Config: Sampling ratio sr.
Input: Seed s, Recall rate r.
Output: Prefix length l.

```
/* (1) Sampling tests with ratio $sr$                                    */
1  $p \longleftarrow sr \cdot \textsc{AssignEnergy}(s)$
2  inputCache $\longleftarrow [\emptyset \dots \emptyset]_p$
3  for $i$ from $0$ to $p$ do
4      inputCache[$i$] $\longleftarrow \textsc{Mutate}(s)$

/* (2) Identify if each full execution is interesting or
   not, and record average execution length.                             */
5  arrFull, avgLen $\longleftarrow$ $\textsc{BatchPrefixExecution}$(fullMap, inputCache,
    $-1$)

/* (3) Binary search the minimal prefix length that reaches
   target recall $r$.                                                    */
6  left $\longleftarrow 0$
7  right $\longleftarrow$ avgLen
8  $l' \longleftarrow$ right, $l \longleftarrow -1$
9  while left $<$ right do
10     $\textsc{InitializeHashMap}$(prefixMap)
11     arrPrefix, _ $\longleftarrow$ $\textsc{BatchPrefixExecution}$(prefixMap,
        inputCache, $l'$)
12     if $\textsc{CalculateRecall}$(arrFull, arrPrefix) $\geq r$ then
13         right, $l \longleftarrow l'$
14     else
15         left $\longleftarrow l'$
16     $l' \longleftarrow$ (left $+$ right) / 2
17 return $l$
```

```
Algorithm 4: BatchPrefixExecution
Input: hashMap, inputCache, prefix length $l$.
Output: An array of interestingness arrIntsg, average execution
        length avgLen.
1  $p \longleftarrow$ LengthOf(inputCache)
2  arrIntsg $\longleftarrow [0 \ldots 0]_p$
3  avgLen $\longleftarrow 0$
4  for $i$ from 0 to $p$ do
5      $\Pi^i(l)$, len $\longleftarrow$ PrefixExecution(inputCache[$i$], $l$)
6      avgLen $\longleftarrow$ avgLen + len
7      $h_i \longleftarrow$ Hash($\Pi^i(l)$)
8      if hashMap[$h_i$] == 0 then
9          hashMap[$h_i$] $\longleftarrow 1$
10         arrIntsg[$i$] $\longleftarrow 1$
11     else
12         arrIntsg[$i$] $\longleftarrow 0$
13 avgLen $\longleftarrow$ avgLen / $p$
14 return arrIntsg, avgLen
```

2. Before performing prefix length search, the interestingness of each full execution is learned via BatchPrefixExecution with prefix length $l = -1$ (recall `__afl_prefix_len=-1` in Section 3.2.2). The hashmap `fullMap` records hashes of all visited executions. `arrFull` is a binary array of size p, where `arrFull`$[i] = 1$ indicates `inputCache`$[i]$ being interesting. `avgLen` is the average execution length.

3. From 0 to `avgLen`, we binary search for the minimal prefix length that reaches the target recall r. During each round of search, `prefixMap` is initialized first. With the same prefix length l', we learn the interestingness of each execution prefix via BatchPrefixExecution. The binary array `arrPrefix` stores such information just as `arrFull`. The function CalculateRecall calculates the recall rate of l'. The final prefix length l will be updated to l' when its recall rate is $\geq r$.

Function BatchPrefixExecution is described in Alg. 4. Each of the tests (line 4) first gets the execution prefix of length l (full execution when $l = -1$). The hash of the execution prefix is calculated (line 7) to index the given hashmap (line 8). If the indexed entry is 0, *i.e.*, this is a new prefix, mark it as seen (line 9) and set `arrIntsg`$[i]$ to 1 (line 10). Otherwise, set `arrIntsg`$[i]$ to 0, *i.e.*, non-interesting (line 12). The variable `avgLen` accumulates execution length (line 6) and reports the average value finally (line 13). The returned array `arrIntsg` records whether execution prefixes of tests are interesting or not.

Function CalculateRecall calculates the recall as follows:

$$r = \frac{\sum_{i=0}^{p-1}(\texttt{arrFull}[i] \land \texttt{arrPrefix}[i])}{\sum_{i=0}^{p-1}\texttt{arrFull}[i]}.$$

Intuitively, this formula calculates to what percentage of the 1's in `arrFull` are also marked as 1 in `arrPrefix`.

3.3 EVALUATION

This section details our extensive evaluation of PGE to demonstrate its effectiveness in improving fuzzing performance. Our evaluation will answer the seven research questions as shown below. The first three RQs are module-only evaluations on the key component PrefixLengthSearch, which adds extra fuzzing overhead. The rest of the RQs focus on PGE's overall performance.

RQ1. Accuracy of prefix length estimation at different sampling ratios,

RQ2. Overhead of the PrefixLengthSearch module,

RQ3. Distributions of prefix length on different recall settings,

RQ4. Early terminated tests as a percentage of all tests,

RQ5. Ratio of executing interesting tests to all full executions,

RQ6. Effectiveness of fuzzing in terms of bug finding and

RQ7. Effectiveness of fuzzing in terms of code coverage.

Experimental Setup. We used AFL++ 4.01c [35], the latest version at the time of writing, as the reference fuzzer to study the benefits of PGE. We refer to our augmented AFL++ as AFL++-PGE. We chose to use the most recent fuzzing benchmark Magma v1.2 [52], the latest version at the time of writing. Magma consists of 21 programs from nine popular real-world projects,

Table 3.3: Magma targets.

Target	Drivers	Version	File type
libpng	libpng_read_fuzzer	1.6.38	PNG
libsndfile	sndfile_fuzzer	1.0.31	Audio
libtiff	tiff_read_rgba_fuzzer, tiffcp	4.3.0	TIFF
libxml2	xml_read_memory_fuzzer,xmllint	2.9.12	XML
lua	lua	5.4.3	LUA
openssl	asn1, asn1parse, bignum, server, client, x509	3.0.0	*Binary blobs*
php	json, exif, parser, unserialize	8.1.0alpha3	*Various*
poppler	pdf_fuzzer, pdfimages, pdftoppm	21.07.0	PDF
sqlite3	sqlite3_fuzz	3.37.0	SQL

which were selected for their diverse functionalities. Table 3.3 details these targets. Due to our implementation limitation, we cannot support persistent fuzzing, and thus, all persistent targets such as pdf_fuzzer are using AFL driver with $N = 1$.

To test PGE's effectiveness against other fuzzing throughput boosters, we compare it with HeXcite [106], the state-of-the-art coverage-guided tracer for coverage-guided fuzzers. HeXcite improves fuzzing throughput by restricting instrumentation overhead to coverage-increasing tests only. It extends Untracer [105] with the support of edge coverage.

We performed our experiments on two servers, each equipped with an AMD Ryzen Threadripper 3990X 64-Core 2.9GHz CPU and 256 GB RAM, and running Ubuntu 20.04.3 LTS. Following Klees *et al.*'s [65] standard we performed each fuzzing campaign for 48 hours and repeated it 12 times.

3.3.1 *RQ1: Accuracy of Prefix Length Estimation at Different Sampling Ratios.*

Given a target recall, PREFIXLENGTHSEARCH simulates fuzzing with sampling data points to estimate an appropriate prefix length that can achieve

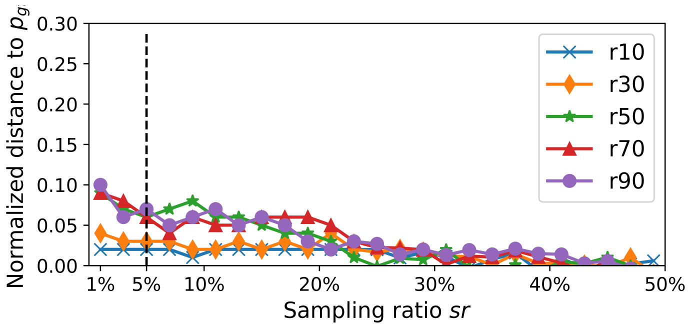


Figure 3.6: Estimation accuracy of different sampling ratios.

it. To understand what sampling ratio is required for an accurate estimate for each program and a target recall, we

1. randomly select a seed,
2. set the sampling ratio $sr = 100\%$ to get the ground truth prefix p_{gt},
3. set sr again from 1% to 50% with step 2% and estimate the prefix p_i, and
4. for each p_i, calculate its normalized distance to p_{gt} with $d_i = \frac{|p_i - p_{gt}|}{p_{gt}}$.

We select 6 recalls spanning evenly from 0% to 100%, namely 10%, 30%, 50%, 70%, and 90%. Each experiment is repeated 12 times, and we report the averaged results across programs in Figure 3.6. A higher sampling ratio indicates a more precise approximation of prefix length but a higher overhead due to the increased number of tests. When $sr > 5\%$, the distance to the ground truth prefix decreases slowly for all recalls. For example, extending sr from 5% to 20% only reduces the distance to less than $0.05 \cdot p_{gt}$. Since higher sr indicates higher overhead, we conclude that $sr = 5\%$ is a good balance between the estimation accuracy and overhead. In our implementation of AFL++-PGE, we set $sr = 5\%$ as default.

In order to understand whether or not PGE can maintain target recalls with a sampling rate 5%, for each target recall, we reuse the above experiment meta-data as follows:

1. estimate the prefix length p with 5% sampled executions,
2. for all executions, count the number of pattern-interesting full executions N_{full},

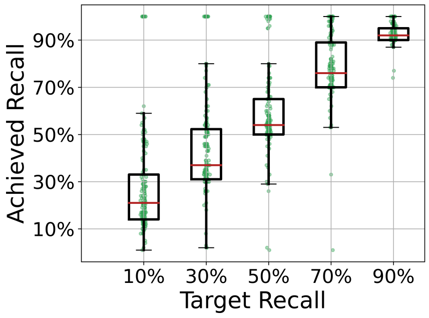


Figure 3.7: Achieved recall vs. target recall with sampling ratio 5%.

3. for all executions, collect their prefixes of length p and then count the number of interesting prefixes N_{pre},
4. and calculate the true achieved recall rate $\frac{N_{pre}}{N_{full}}$.

Figure 3.7 shows the distribution of achieved recalls on all programs. The X-axis refers to the target recall set for PGE, and the Y-axis is the achieved recall. Each box shows the distribution of achieved recalls on all programs. We can find that for every target recall, PGE with 5% sampling rate can always achieve it in more than 75% of cases. The short length of each box indicates that the achieved recalls are concentrated around the target recalls. Overall, PGE with a 5% sampling rate has successfully searched for proper prefixes.

3.3.2 *RQ2: Overhead of Prefix Length Search*

Since *prefix length search*, *i.e.*, function PrefixLengthSearch, brings extra overhead to the fuzzing loops, we first need to understand its cost in time against the overall fuzzing overhead. If AFL++-PGE took much time in searching for a prefix length, even though it could benefit from prefix execution, its overall performance would still be severely affected. To answer this question, we recorded time spent executing PrefixLengthSearch during the fuzzing campaigns.

AFL++-PGE needs an extra input parameter, recall rate r, for guiding PrefixLengthSearch. We select 6 different recalls spanning evenly from 0% to 100%, namely $r = 10\%, 30\%, 50\%, 70\%,$ and 90%. For each target program, we ran AFL++-PGE with different recalls and averaged their PrefixLength-

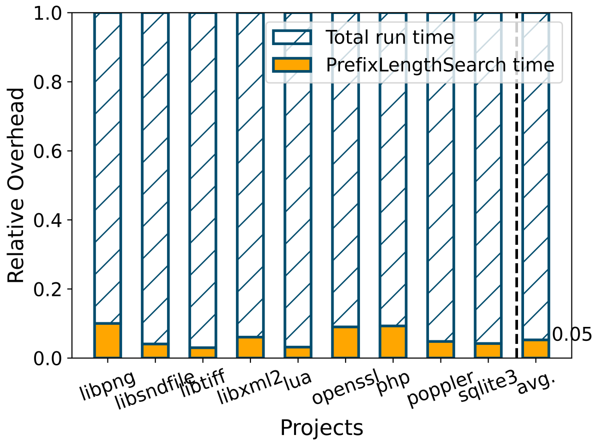


Figure 3.8: Per-program relative overhead of PrefixLengthSearch in 48h.

Search time. The final averaged overhead is reported in Figure 3.8. The last bar in Figure 3.8 shows that during 48 hours of fuzzing campaign, the average time spent executing PrefixLengthSearch is less than 5%, approximately two hours. For most programs, PrefixLengthSearch overhead is negligible. Projects `libpng`, `openssl`, and `php` have relatively larger overhead but all below 10%. Overall, the extra overhead from PrefixLengthSearch did not hinder AFL++-PGE's efficiency.

3.3.3 *RQ3: Distributions of Prefix Length on Different Recall Settings*

Prefix length is crucial for the efficiency of PrefixExecution. Intuitively, a shorter prefix length means a faster PrefixExecution. Understanding what the distributions of prefix length would be on different recall settings is thus essential to understanding the overall performance of AFL++-PGE.

For each trial on a program, we logged the searched prefix lengths and execution lengths. Note that, for some seeds, PrefixLengthSearch may return -1, meaning that there is no effective prefix length, and the fuzzer should continue with the original FullExecution for this seed. This is because sometimes even the maximal prefix (*i.e.*, average full execution length) cannot achieve the target recall. For seeds with effective prefix length, we average their relative prefix lengths *w.r.t.* execution lengths on

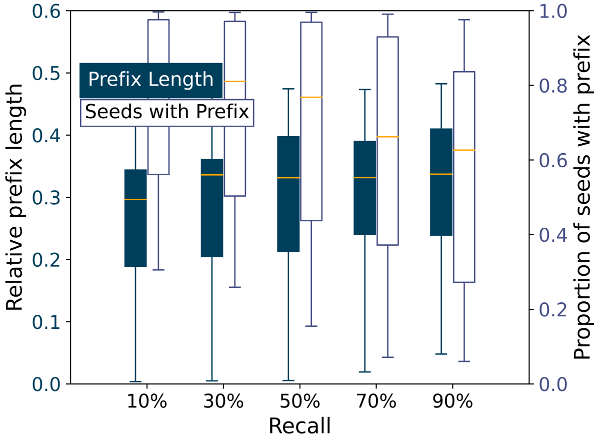


Figure 3.9: Distributions of prefix lengths and proportion of seeds with prefix.

the program-level granularity. Distributions of relative prefix lengths are shown as the solid boxes in Figure 3.9. As expected, a higher recall needs larger prefix lengths. We also show the proportion of seeds with effective prefix length as the hollow boxes in Figure 3.9. Inversely, the number of seeds with effective prefix length is less for a higher recall. For instance, when targeting the 50% recall rate, on average, more than 45% seeds can be run with PrefixExecution, among which the searched prefix lengths are approximately one-third to full execution lengths. Although lower recall rates come with shorter prefix lengths, they, by design, have higher probabilities of missing interesting tests. Section 3.3.6 and 3.3.7 will show the trade-off.

3.3.4 *RQ4: Early Terminated Tests as A Percentage of All Tests*

With prefix-guided execution, a fuzzer saves execution time by only partially executing tests. We now aim to answer how many tests would be early terminated by AFL++-PGE. An early terminated test means that it has been only partially executed. Figure 3.10 shows the mean ratios of early terminated executions per benchmark. We can observe that lower recall settings have a relatively higher percentage of early terminated tests. Intuitively, a lower recall allows PGE to search for a shorter prefix, which is

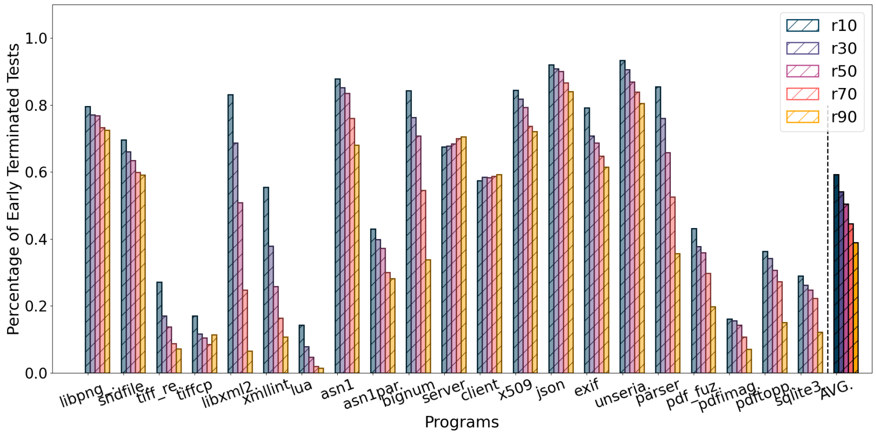


Figure 3.10: Percentage of early terminated executions in AFL++-PGE.

less sensitive compared to longer prefixes and thus results in more early terminated tests. AFL++-PGE with recall 10% early terminated more than 90% tests on programs "asn1", "json", and "unserialize". As will be shown in Section 3.3.6 and 3.3.7, such a low recall, however, enables PGE to discover more bugs and achieve higher coverage on these benchmarks. On average, AFL++-PGE with different recalls have 40% to 60% of tests being early terminated. For the fully executed tests, the majority of them are from seeds where PGE does not find effective prefixes.

3.3.5 *RQ5: Ratio of Executing Interesting Tests to All Full Executions*

We have shown in Section 3.1 that most of the fuzzer-generated tests are not interesting, *i.e.*, in the first hour of fuzzing, only 0.46% tests are coverage-interesting. This observation motivated our PGE design. Ideally, PGE should be more concentrated than a vanilla fuzzer on executing interesting tests. To measure such concentration, we define the interesting ratio as follows:

$$r = \frac{\text{\#coverage-increasing tests}}{\text{\#fully executed tests}}.$$

For AFL++, all tests are fully executed. In AFL++-PGE, only partial tests are fully executed since many of the generated tests are early terminated by PGE. We calculate the interesting ratios for each fuzzer on each program

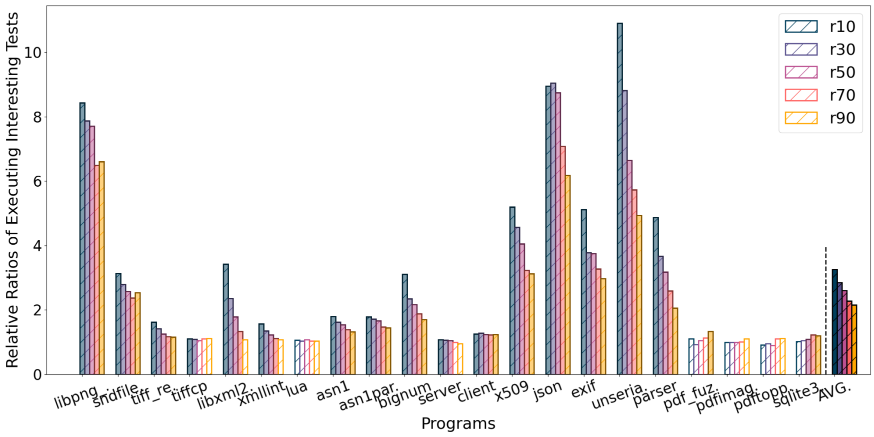


Figure 3.11: AFL++-PGE's ratio of executing coverage-increasing tests to all full executions relative to AFL++. Each ratio is calculated by #coverage-interesting tests / #full executions. Each solid bar indicates that the corresponding AFL++-PGE has a significantly higher ratio than AFL++ (*i.e.*, Mann-Whitney U test $p < 0.05$).

and average the ratios across all trials. We use Mann-Whitney U-*test* to measure the statistical significance compared to AFL++. Figure 3.11 shows the mean relative ratios of AFL++-PGE to AFL++ on each program. We can see that on 18 out of 21 programs, at least one of the AFL++-PGE has a significantly higher interesting ratio than the vanilla AFL++. On 12 out of 21 programs, all AFL++-PGE have significantly higher interesting ratios. For programs "libpng_read_fuzzer", "x509", "json", "exif", "unserialize", and "parser", AFL-PGE have 3x to 11x higher interesting ratios.

Note that, the interesting ratios do not directly reflect the effectiveness of the fuzzers. For example, although AFL++-PGE-r90 has a relatively low ratio on "sqlite3", as shown in Section 4.6, AFL++-PGE-r90 can find 3 bugs on it while the vanilla AFL++ can only find 1 bug. The main reason for the relatively low interesting ratios of AFL++-PGE on some programs is that after 48 hours of fuzzing, AFL++-PGE saturates on some programs, which leads to a significant increase in the total number of full executions, while the increase in coverage is subtle due to saturation, resulting in a less significant interesting ratio r. As shown in Section 3.3.3, PGE can not always find an effective prefix. For these programs, all PGE instances have turned back to the normal fuzzing procedure on some seeds, which results in the same ratios on these seeds.

3.3.6 *RQ6: Bug-Finding Evaluation*

The ability to discover bugs is the golden metric for fuzzing performance. We evaluate the bug detection capability of each fuzzer with *bug survival time*, which is proposed in the Magma. This metric uses the time required to trigger a bug to measure bug-finding speed. Due to the highly stochastic nature of fuzzing, the time-to-bug might differ dramatically across repeated attempts. To mitigate the high variations in bug discovery time, Magma recommends the use of *survival analysis* to infer how long a bug "survives" in a fuzzing campaign. It adopts Wagner's approach [144] and uses the Kaplan-Meier estimator [64] to estimate a bug's survival time, *i.e.*, staying undiscovered, within a given time (48 hours in our evaluation). A shorter survival time means better fuzzing performance. We also use the *log-rank test* [99, 53] to compare bug survival times statistically. The *log-rank test'* p-value<0.05 implies statistical significance. Table 3.4 summarizes the bugs found in Magma and the mean survival time. We omit programs when none of the fuzzers were able to find any bugs in them and bugs when all fuzzers were able to find them without statistical significance. Due to HeXcite's lack of support for the persistent mode that most programs use, we were only able to run it on five programs.

Versus AFL++: In total, AFL++ was able to find 25 bugs, and all AFL++-PGE together uniquely discovered 10 more. Specifically, AFL++-PGE with recall 10%, 30%, 50%, 70%, and 90% covered 4, 6, 5, 2, and 7 more bugs than vanilla AFL++. Of all these bugs[4], these AFL++-PGE were faster than AFL++, respectively, on 8, 11, 9, 9, and 11 of them, and slower, respectively, on 5, 6, 4, 3, and 2 of them. Overall, AFL++-PGE-r90 was the best in terms of bug-finding ability.

An interesting fact is that a very low recall still has a strong ability to discover bugs. For instance, AFL++-Pge with recall 30Intuitively, a lower recall indicates that the searched prefix length has a higher probability of missing interesting tests. However, on one hand, we observed from Magma's intermediate logs that nearly all bugs, if being triggered, would be triggered more than once. Suppose that for some bug, multiple triggering tests are

4 When we compare AFL++-PGE-rXX with AFL++, only bugs triggered by at least one of these two fuzzers are included.

Table 3.4: Mean survival time in 48 hours. Bugs that have never been found are marked by "⊤". Survival times either statistically better than AFL++ (p-val < 0.05) or uniquely identified by AFL++-PGE are highlighted in green. "✗" means the program is incompatible with the fuzzer.

			AFL++-PGE											
Program	Bug	AFL++	r10	*p*-val	r30	*p*-val	r50	*p*-val	r70	*p*-val	r90	*p*-val	HeXcite	*p*-val
libpng	PNG001	⊤	⊤	-	⊤	-	48h	-	⊤	-	45h	-	✗	-
	PNG007	31h	19h	0.08	6h	0.00	8h	0.00	10h	0.00	9h	0.00	✗	-
libsndfile	SND001	0.4h	0.2h	0.14	0.2h	0.43	0.3h	0.62	0.3h	0.58	0.4h	0.77	47h	0.00
	SND005	0.9h	1h	0.60	2h	0.73	1.0h	0.54	0.5h	0.01	0.7h	0.22	0.8h	0.48
	SND024	0.6h	0.4h	0.01	0.2h	0.00	0.2h	0.00	0.3h	0.00	0.4h	0.05	⊤	-
libtiff- tiff_read _rgba	TIF002	38h	47h	0.05	⊤	-	⊤	-	45h	0.17	47h	0.04	✗	-
	TIF008	42h	⊤	-	47h	0.23	⊤	-	⊤	-	⊤	-	✗	-
libtiff -tiffcp	TIF006	22h	32h	0.13	38h	0.02	32h	0.21	39h	0.01	36h	0.08	18h	0.81
	TIF007	0.1h	0.1h	0.45	0.1h	0.17	0.1h	0.64	0.0h	0.88	0.0h	0.40	32h	0.00
	TIF014	2h	1h	0.22	2h	0.02	2h	0.16	1h	0.29	2h	0.27	⊤	-
libxml2- xml_read _memory	XML001	1h	0.4h	0.00	0.8h	0.01	2h	0.22	1h	0.89	2h	0.47	✗	-
	XML009	2h	1h	0.13	3h	0.67	1h	0.02	0.8h	0.01	2h	0.77	✗	-
	XML012	⊤	⊤	-	⊤	-	48h	-	⊤	-	48h	-	✗	-
libxml2 -xmllint	XML001	44h	⊤	-	48h	0.04	47h	0.15	47h	0.29	45h	0.55	⊤	-
	XML002	⊤	⊤	-	46h	-	⊤	-	⊤	-	⊤	-	⊤	-
	XML009	1.0h	3h	0.03	2h	0.18	0.9h	0.86	2h	0.22	3h	0.03	⊤	-
openssl -asn1	SSL001	⊤	10h	-	5h	-	4h	-	10h	-	4h	-	✗	-
openssl -server	SSL002	0.2h	0.2h	0.00	0.2h	0.00	0.2h	0.00	0.2h	0.00	0.2h	0.00	✗	-
	SSL020	45h	28h	0.02	40h	0.32	41h	0.32	29h	0.02	30h	0.04	✗	-
php	PHP009	1h	13h	0.00	14h	0.00	6h	0.02	10h	0.00	3h	0.57	✗	-
poppler -pdfimages	PDF008	46h	⊤	-	44h	0.95	⊤	-	47h	0.95	45h	0.95	⊤	-
	PDF014	⊤	⊤	-	46h	-	⊤	-	⊤	-	45h	-	⊤	-
	PDF018	5h	36h	0.00	27h	0.00	25h	0.00	27h	0.00	15h	0.16	⊤	-
	PDF021	43h	40h	0.40	39h	0.57	40h	0.63	⊤	-	45h	0.93	⊤	-

Table 3.5: (Continuous) of Table 3.4

			AFL++-PGE					
Program	Bug	AFL++	r10 *p*-val	r30 *p*-val	r50 *p*-val	r70 *p*-val	r90 *p*-val	HeXcite *p*-val
poppler -pdftoppm	PDF010	2h	1h 0.11	1h 0.21	1h 0.17	0.8h **0.01**	1h 0.12	31h 0.10
	PDF011	45h	46h 0.95	46h 0.62	47h 0.95	44h 0.95	41h 0.55	37h **0.01**
	PDF018	11h	36h **0.00**	28h **0.02**	36h **0.02**	17h 0.71	18h 0.54	⊤ -
	PDF019	⊤	46h -	45h -	47h -	⊤ -	⊤ -	⊤ -
poppler -pdf_fuzzer	PDF011	⊤	46h -	⊤ -	⊤ -	⊤ -	44h -	✗ -
	PDF014	⊤	⊤ -	48h -	⊤ -	⊤ -	⊤ -	✗ -
	PDF018	11h	42h **0.00**	37h **0.00**	44h **0.00**	26h 0.08	21h 0.14	✗ -
	PDF021	46h	41h 0.32	⊤ -	47h 0.55	43h 0.86	44h 0.86	✗ -
sqlite3	SQL012	47h	⊤ -	⊤ -	47h 0.95	⊤ -	48h 0.95	✗ -
	SQL013	⊤	⊤ -	46h -	47h -	⊤ -	48h -	✗ -
	SQL020	⊤	45h -	44h -	⊤ -	41h -	46h -	✗ -

generated during fuzzing. Even if every single test has a low probability of being executed, the probability that this bug is triggered at least once is still high according to the chain rule [128] in probability theory. But the triggering time might be delayed. For example, AFL++-PGE-r10 triggered "PHP009" much later than many others. On the other hand, Section 3.3.4 has shown that AFL++-PGE-r10 has the largest fuzzing throughput, which allows it to explore some new paths within the time constraint.

Versus HeXcite: HeXcite has the fastest discovery speed on "PDF011 (pdftoppm)". For the rest of the 5 bugs it discovered, AFL++-PGE-r90 is faster on 3 of them. On the 5 programs it supports, it missed 8 bugs compared to AFL++, and AFL++-PGE-r90 found 9 more bugs than it with statistical significance. Intuitively, HexCite should be able to boost fuzzing efficiency since it improves overall throughput. However, it does not guarantee that all critical edges that are important to path exploration will be traced. As its empirical evaluation revealed, HexCite only tracks 89% of such edges, which inevitably limits its path-discovering capability. Moreover, hit count coverage in HexCite is limited to loops only. Hit counts refer to edge execution frequencies. Due to its high overhead in tracking all hit counts, HexCite focuses only on loops, which further hinders the exhaustiveness of its exploration.

3.3.7 *RQ7: Coverage Evaluation*

Code coverage is a common and popular evaluation metric for fuzzing when ground-truth bugs are not available. Previous work [65, 80] argued that higher coverage does not necessarily indicate better bug-finding capability. Nevertheless, for the thoroughness of comparison, we report branch coverage achieved by each fuzzer. We measure each trial's edge coverage with `afl-showmap` utility and compute the average coverage across all trials. We use Mann-Whitney *U-test* to measure the statistical significance compared to AFL++. Table 3.6 summarizes the coverage results, where statistically higher coverage than AFL++ (*i.e.*, p-value < 0.05) is highlighted in green.

Versus AFL++: As Table 3.6 shows, of the total 21 programs, AFL++-PGE with recall 10%, 30%, 50%, 70%, and 90% achieve statistically higher coverage than AFL++ respectively on 10, 10, 10, 10 and 11 of them, and the same on the left. Intuitively, a lower recall indicates that PGE would have a higher probability of missing interesting tests. However, all PGE achieve nearly the same coverage at the end of 48h fuzzing. The main reason is that all PGE have become saturated on these benchmarks. No matter how many new tests are explored, the overall improvements in coverage will be subtle. Thus, although PGE boosts the overall fuzzing speed substantially, it results in relatively small coverage improvements. For example, for recalls 10% and 30%, although they can explore more tests as shown in Section 3.3.4, their coverage performance does not surpass others. The highest recall 90% performed the best, which is aligned with its strongest bug-finding capability as shown in Section 3.3.6.

Higher coverage does not necessarily mean a stronger bug-finding capability. For instance, as shown in Section 3.3.6, AFL++-PGE-r50 statistically found more bugs than AFL++ in `libpng` and `sqlite3`, however, it achieves lower coverage than AFL++. As the bug-finding capability is the most important measure in fuzzing, PGE's significance in this measurement has shown its effectiveness in boosting fuzzing performance.

Versus HeXcite: For the five programs that HeXcite supports, it has statistically lower coverage than AFL++ on 4 of them and never outperforms AFL++. The best performing AFL++-PGE-r90 achieves higher coverage on

Table 3.6: Edge coverage (%) achieved by fuzzers. Coverages statistically higher than AFL++ are highlighted in green. "✗" means the program is incompatible with the fuzzer.

		AFL++-PGE					
Program	AFL++	r10$_{\text{MWU}}$	r30$_{\text{MWU}}$	r50$_{\text{MWU}}$	r70$_{\text{MWU}}$	r90$_{\text{MWU}}$	HeXcite$_{\text{MWU}}$
libpng_read_fuzzer	24.6	24.3$_{0.77}$	24.6$_{0.58}$	24.3$_{0.76}$	24.6$_{0.45}$	24.4$_{0.67}$	✗$_{-}$
sndfile_fuzzer	22.2	22.1$_{0.94}$	22.1$_{0.89}$	22.2$_{0.61}$	22.2$_{0.57}$	22.3$_{0.22}$	17.3$_{0.00}$
xml_read_memory	18.5	18.2$_{1.00}$	18.3$_{1.00}$	18.4$_{1.00}$	18.5$_{0.59}$	18.6$_{0.00}$	✗$_{-}$
xmllint	16.3	15.9$_{1.00}$	16.1$_{1.00}$	16.2$_{1.00}$	16.2$_{1.00}$	16.2$_{0.96}$	11.3$_{0.00}$
lua	81.5	80.6$_{0.93}$	80.9$_{0.94}$	81.6$_{0.27}$	81.4$_{0.56}$	81.3$_{0.60}$	✗$_{-}$
asn1	10.5	11.7$_{0.00}$	11.8$_{0.00}$	11.8$_{0.00}$	11.8$_{0.00}$	11.8$_{0.00}$	✗$_{-}$
asn1parse	1.4	2.0$_{0.00}$	2.0$_{0.00}$	2.0$_{0.00}$	2.0$_{0.00}$	2.0$_{0.00}$	✗$_{-}$
bignum	1.2	1.9$_{0.00}$	1.9$_{0.00}$	1.9$_{0.00}$	1.9$_{0.00}$	1.9$_{0.00}$	✗$_{-}$
server	15.3	16.9$_{0.00}$	16.9$_{0.00}$	16.9$_{0.00}$	16.9$_{0.00}$	16.9$_{0.00}$	✗$_{-}$
client	15.1	17.1$_{0.00}$	17.1$_{0.00}$	17.1$_{0.00}$	17.1$_{0.00}$	17.1$_{0.00}$	✗$_{-}$
x509	11.6	12.4$_{0.00}$	12.4$_{0.00}$	12.4$_{0.00}$	12.4$_{0.00}$	12.4$_{0.00}$	✗$_{-}$
tiff_read_rgba	36.0	32.6$_{1.00}$	34.0$_{1.00}$	34.1$_{1.00}$	34.7$_{1.00}$	34.9$_{1.00}$	✗$_{-}$
tiffcp	41.0	39.8$_{1.00}$	40.1$_{1.00}$	39.6$_{1.00}$	40.4$_{0.89}$	40.4$_{0.91}$	41.0$_{0.23}$
json	0.7	1.5$_{0.00}$	1.5$_{0.00}$	1.5$_{0.00}$	1.5$_{0.00}$	1.5$_{0.00}$	✗$_{-}$
exif	0.9	1.7$_{0.00}$	1.8$_{0.00}$	1.8$_{0.00}$	1.8$_{0.00}$	1.8$_{0.00}$	✗$_{-}$
unserialize	1.0	1.8$_{0.00}$	1.8$_{0.00}$	1.8$_{0.00}$	1.8$_{0.00}$	1.8$_{0.00}$	✗$_{-}$
parser	4.9	5.3$_{0.00}$	5.4$_{0.00}$	5.5$_{0.00}$	5.5$_{0.00}$	5.5$_{0.00}$	✗$_{-}$
pdf_fuzzer	38.8	38.3$_{1.00}$	38.4$_{1.00}$	38.7$_{0.86}$	38.7$_{0.59}$	38.7$_{0.63}$	✗$_{-}$
pdfimages	45.7	44.8$_{1.00}$	45.1$_{1.00}$	45.2$_{1.00}$	45.5$_{0.93}$	45.7$_{0.22}$	36.7$_{0.00}$
pdftoppm	39.2	38.6$_{1.00}$	38.7$_{1.00}$	39.0$_{0.90}$	39.0$_{0.90}$	39.0$_{0.83}$	31.8$_{0.00}$
sqlite3_fuzz	42.6	40.3$_{1.00}$	40.3$_{1.00}$	41.3$_{0.98}$	42.4$_{0.53}$	43.3$_{0.11}$	✗$_{-}$

Table 3.7: Bugs detected by AFL++ and AFL++-PGE-r90.

			AFL++			AFL++-PGE-r90		
Target	File Type	Version	Found	New	Fixed	Found	New	Fixed
ffmpeg	Video	4.4	1	0	0	1	0	0
cflow	C	1.6.92	0	0	0	1	0	0
mp42aac	MP4	1.6.0	2	0	0	4	2	0
objdump	Binary	2.36.1	8	8	8	10	10	9
readelf	Binary	2.36.1	5	5	5	5	5	5
nm-new	Binary	2.36.1	8	7	7	12	11	10
Total			24	20	20	**33**	**28**	**24**

4 of them. Although HeXcite is able to improve fuzzing throughput by design, as has been analyzed in Section 3.3.6, it trades its efficacy for speed by discarding many instrumentations. As reported in HeXcite, it is more significant in black-box settings and achieves nearly equivalent or worse performance in grey-box settings. Our evaluation confirmed this fact.

3.3.8 *Finding Bugs in Latest Applications*

Evaluations on the Magma benchmark programs have already shown the superiority of AFL++-PGE. In order to test if AFL++-PGE is able to boost AFL++'s bug-finding capability on real-world applications, we selected 6 up-to-date programs with diverse types. All programs have been well-fuzzed in the fuzzing community [145, 6, 58, 21]. We used their official test suites as the initial seeds for `ffmpeg`, `objdump`, `readelf`, and `nm-new`. Seeds for the remaining two programs are from UNIFUZZ [80]. We set recall for AFL++-PGE to 90% as suggested by our evaluations on Magma. All experiments were done on the same environments as discussed in Section 3.3 with a timeout of 48 hours. Table 3.7 reports the number of bugs found by each fuzzer. We manually inspected and triaged all unique crashes to identify unique bugs. In summary, AFL++-PGE-90 discovered more bugs than AFL++ in 4 out of the 6 programs. All bugs found by AFL++ were also discovered by AFL++-PGE-r90, which means that PGE did not miss any bugs in these real-world applications. AFL++-PGE-r90 in total discovered 28 previously unknown bugs, 40% more than AFL++. We reported these bugs to the developers, and 24 of them have already been fixed. We can thus conclude that PGE significantly improves AFL++'s performance in finding bugs in real-world applications.

3.3.9 *Discussion*

Impact of recall in PGE. As can be learned from Sections 3.3.3 and 3.3.4, a lower recall rate means a shorter prefix and higher execution speed but also means a higher probability of missing interesting tests. As a result, we should avoid selecting a recall that is too small. On the other hand, experimental results in Section 3.3.7 and 3.3.6 showed that higher recall did not always mean more effective performance. For example, AFL++-PGE-r90 performed better than AFL++-PGE-r50 in both code coverage and bug survival time. Although AFL++-PGE-r10 and r30 did not outperform AFL++-PGE-r90 overall, they found additional bugs in `xmllint` and `pdftoppm`. In summary, it is a trade-off to balance execution speed and bug-triggering probability. We conclude that the higher execution speed of relatively low recall compensated its low bug-triggering probability. A proper target recall in PGE should be neither too large nor too small. Recall 90% is the best-performing one considering both of our measures and can be a good choice in practice.

Orthogonality of PGE to various CGF efforts. At a high level, PGE is a surrogate module for full executions and does not affect other parts of a coverage-guided fuzzer. Algorithm 2 in Section 3.2.1 has shown the algorithmic sketch for a coverage-guided grey-box fuzzer (CGF) and our PGE extensions to it. Various CGF efforts try to improve different aspects of the fuzzing process and share the same CGF workflow. For instance, seed scheduling schemes such as AFLFast [10] optimize the SelectSeed (line 2) and AssignEnergy (line 4) functions. Mutator scheduling methods such as MOPT [95] improve the Mutate function. AFL++ [34] integrates many advances into AFL while still maintaining the same CGF workflow. PGE does not alter any of these optimized functions and only conditions full executions *w.r.t.* their prefixes. PGE is therefore orthogonal to these efforts. Our extensive evaluation has shown a significant performance improvement of PGE to AFL++. We believe the integration of PGE to other fuzzers will boost their performance as well. When integrating PGE into a new fuzzer, the main efforts are to update two components coordinately: For

`PrefixExecution`, if the new fuzzer does not use AFL-style instrumentation, one needs to add a global counter and a guard for prefix collection in the instrumentation code; For `PrefixLengthSearch`, mutation strategies should be aligned with the new fuzzer.

Support of stateful fuzzing. It is crucial to find security vulnerabilities in stateful and persistent targets such as network services. Stateful fuzzing, however, is difficult. Vanilla CGF fuzzers like AFL++ are limited in their abilities to support stateful targets like network services. Since PGE is built atop AFL++, it inherits this limitation. There is some interesting work that attempts to address this limitation. One representative effort is AFLNet [120], which augments AFL to make it state-aware. In principle, PGE could be integrated into it since AFLNet treats a message sequence as an input seed in a normal fuzzing scenario while still maintaining state information. Early termination in AFLNet would not lose the target state.

Compatibility to other fuzzing boosting schemes. An interesting and orthogonal boosting approach for fuzzing is checkpoint-based resume [135]. The key insight is that kernel fuzzers frequently execute similar test cases with the same prefixes. To avoid redundant prefix executions, checkpoints are used to save states for hot prefixes. All future test cases with prefixes being cached can then resume from checkpoints. In principle, this scheme could be augmented with PGE. The augmented fuzzer can start collecting prefixes from a resumed checkpoint and terminate a subsequent execution when its prefix is uninteresting. Since this scheme is applied in kernel fuzzing and PGE is initially designed for application fuzzing, it remains challenging and unknown how PGE would perform on top of it.

Limitations and future work. PGE opens up a new and orthogonal research direction for improving fuzzing efficiency by early terminating executions. In our current design, we make use of the coverage information from executions as the indicator for early termination. Although it has shown a superior performance, our current PGE cannot use a hundred percent recall since it would require a longer, thus less effective prefix. This is mainly due to the limited expressiveness of the coverage pattern as the indicator for early termination. It would be an exciting and interesting future work to explore other more expressive indicators such as data flows [39] and variable states [33]. Such indicators may have the potential to predict an execution's coverage increasingness in a much earlier stage and thus need a shorter prefix.

3.4 THREATS TO VALIDITY

Here, we discuss potential threats to the validity of our results and conclusions.

Threats to external validity. One threat to external validity is the benchmarking programs we used for our evaluation. To reduce this threat, we chose the Magma benchmark, which consists of 21 programs from nine real-world projects. All of them have been widely used for evaluating fuzzers' performance in the fuzzing community. These programs were carefully selected by the Magma authors according to their diversity in functionality. Another threat to external validity is the randomness in fuzzing. Due to the highly stochastic nature of fuzzing, different trials on the same benchmark may differ significantly. To deal with this issue, we repeated all our experiments 12 times and used the log-rank test and Mann-Whitney U-*test* to draw statistically significant conclusions.

Threats to conclusion validity. This threat to validity relates to the reliability of the chosen measurements and the used statistical tests. We measured the significance of our PGE with bug-finding capability and edge coverage. Both measurements are widely used in the fuzzing community. For the adopted statistical tests, we followed many existing fuzzers.

3.5 RELATED WORK

Coverage-guided Grey-box Fuzzing. Since the success of AFL, there is a large body of work that incorporates techniques such as taint analysis [4, 19, 20, 102, 123], symbolic execution [111, 112, 137, 148, 160], static analysis [22, 79, 118], and deep learning [48, 122, 134, 147], to improve the performance of grey-box fuzzing. However, all of them still have to fully execute all generated tests and, to the best of our knowledge, PGE is the first extension to grey-box fuzzing for reducing execution overhead via early termination.

Improving Fuzzing Performance. Since the success of AFL [161], the fuzzing community has seen a broad range of new fuzzer developments. In particular, coverage-guided grey-box fuzzers such as AFL++ [34], AFLFast [9], and AFLGo [8] are the most widely adopted and studied fuzzing techniques. Researchers have also put great efforts into optimizing various aspects of fuzzing, such as seed scheduling [7], mutation strategies [4, 95], and path explorations [58, 137]. In theory, all these improvements are not related to sanitizer-enabled programs and, therefore, are orthogonal to us. In practice, AFL++ has internally integrated many advances such as RedQueen [4] and MOpt [95].

Reducing Coverage Collection Overhead. Some researchers point out that coverage collection in fuzzing brings extra overhead. Untracer [105] and HexCite [106] remove instrumentation code in visited code edges to reduce coverage collection overhead. Zeror [167] shifts between diversely instrumented binaries to achieve low coverage collection overhead on most executions. Odin [149] dynamically recompiles a binary when the instrumentation requirement changes. Because all these approaches need to modify the coverage bitmap, our approach is not compatible with them. INSTRIM [57] utilizes the existence of common program structures to instrument a small fraction of basic blocks for reconstructing coverage. However, it becomes less useful after LLVM incorporates a similar but more efficient functionality in its PCGUARD mode [94]. At the system level, Xu. *et al.* [157] implement three operating primitives specialized for fuzzing to solve the scalability bottlenecks in file I/O and system calls.

These research efforts are related to prefix-guided execution as they all target reducing overhead introduced by fuzzing itself. PGE tackles this problem from an orthogonal and new perspective.

Reachability Prediction in Targeted Grey-box Fuzzing. There is a large body of work [16, 155, 170] on guiding fuzzers toward target locations since AFLGo [8]. Among them, two are particularly related to PGE as they share a similar philosophy. FUZZGUARD [170] trains a deep learning model for each target location and predicts if each test can reach the target location without executing them. It hypothesizes that tests reaching the same target location have common syntactic patterns. Wüstholz. *et al.* [155] present an online static look-ahead analysis to determine if an execution prefix can reach a target location. These two efforts both avoid unnecessary full executions. Their effectiveness is guaranteed by the high concentration of tests in targeted grey-box fuzzing both syntactically and semantically. Although they have different application scenarios from PGE, it is nevertheless interesting future work to explore if deep learning and static analysis are applicable in reducing full executions for general fuzzing.

4 DECOUPLING SANITIZATION FROM FUZZING FOR LOW OVERHEAD

Although Chapter 2 presents CompDiff for detecting more kinds of undefined behaviors, it cannot beat sanitizers in detecting specific undefined behaviors, such as buffer overflow and use after free. At compile-time, when sanitizers are enabled, compilers will heavily instrument the target program to insert various checks. At runtime, violations of these checks will result in program crashes. Fuzzing on sanitizer-enabled programs is thus more effective in discovering software bugs. To date, the most widely-used sanitizers include AddressSanitizer (ASan) [130], UndefinedBehaviorSanitizer (UBSan) [88], and MemorySanitizer (MSan) [136].

Sanitizers, despite their extraordinary bug-discovery capability, have two main drawbacks. First, sanitizers bring significant performance overhead to fuzzing. As our evaluation in Section 4.1.1 will show, ASan, UBSan, and MSan averagely slow down fuzzing speed by a factor of 3.6x, 2.0x, and 46x, respectively. Since fuzzing is computationally intensive, such high sanitizer overheads inevitably impede both the performance of fuzzers and the adoption of sanitizers. Many approaches have been proposed to reduce the run-time overhead of sanitizers. For example, Debloat [165] optimizes ASan checks via sound static analysis. SanRazor [163] removes likely redundant ASan and UBSan checks through dynamic profiling. FuZZan [62] designs dynamic metadata structure to improve the performance of ASan and MSan. Notwithstanding these optimization efforts, the overhead imposed by sanitizers remains considerable. For instance, as our evaluation will show, the state-of-the-art effort, Debloat, can only reduce less than 10% run-time cost of ASan. Moreover, all these schemes require significant modifications to the existing sanitizer code base, which hinders its compatibility with diverse infrastructures.

The second drawback is that some sanitizers are mutually exclusive. For instance, because ASan and MSan maintain the same metadata structure, they can not be used together. Consequently, a fuzzer has to fuzz the ASan-enabled program and MSan-enabled program separately.

Key idea. In this chapter, we introduce a new fuzzing framework that decouples sanitization from the fuzzing loop for acceleration. In the framework, the fuzzer (1) performs fuzzing on a binary that is built normally, *i.e.*, without enabling sanitizers, then (2) selects inputs that have unique execution paths and runs them on the sanitizer-enabled binaries to check if they trigger any bugs. Take the program shown in Figure 1.3 as an example: During fuzzing, when we first encounter an input that has the execution path $\{1 \rightarrow 3 \rightarrow 5 \rightarrow 18\}$, we re-execute the input on the sanitizer-enabled binary. The result is that this input does not trigger any bug. We will not validate all future inputs that have the same execution path on the sanitizer-enabled binary. When we first encounter an input that has the execution path $\{1 \rightarrow 3 \rightarrow 18\}$, similarly, we re-execute the input on the sanitizer-enabled binary. The result is that this input triggers a Use-after-Free bug. As long as only (1) a small fraction of inputs have unique execution paths and (2) the buggy execution reliably has a unique execution path, we can significantly reduce the sanitization overhead during fuzzing. As our evaluation in Section 4.3 will show, only less than 2% of inputs on average have unique execution paths, and more than 96% of buggy executions have a unique execution path.

There is a key challenge in our approach: ***How to efficiently obtain execution path during fuzzing?*** Previous research has demonstrated that obtaining fine-grained execution information like execution path is too costly to be practical [40, 146]. In our design, we tackle this problem by designing an efficient and scalable execution pattern to approximate the execution path. Execution pattern discards the order information in the execution path as a trade-off for efficiency. For instance, for the execution path $\{1 \rightarrow 3 \rightarrow 18\}$, its execution pattern is $\{1, 3, 18\}$ meaning that code regions $1, 3$ and 18 are executed. Although such approximation may cause imprecision in theory, our evaluation will demonstrate that this design is accurate enough to identify unique execution paths.

Our idea is generally applicable to the grey-box fuzzer family. Since AFL++ [34] is the most popular grey-box fuzzer in both academia and industry, we realized our idea on top of it and implemented a tool named SAND. We use 12 real-world programs widely used by the fuzzing community to evaluate SAND. Our evaluation shows that in 24 hours, compared to traditional fuzzing on ASan/UBSan-enabled and MSan-enabled programs, SAND respectively achieves 2.6x and 15x throughput and detects 51% and 242% more bugs. In the meantime, compared to fuzzing on normally built programs, SAND can achieve nearly the same level of throughput while covering 258% more bugs.

Main contributions. In summary, we make the following contributions:

- We identify that bugs are strongly connected with unique execution paths and further design an approximate yet accurate execution pattern to obtain execution path information during fuzzing efficiently.

- We propose a novel fuzzing framework that decouples sanitization from the fuzzing loop by selectively feeding fuzzer-generated inputs into sanitizers.

- We implement our idea in a tool named SAND. We conduct in-depth evaluations to understand its effectiveness in terms of bug-finding, throughput, and coverage.

4.1 OBSERVATION AND ILLUSTRATION

In this section, we first introduce our observations on the high overhead introduced by sanitizers and the rarity of bug-triggering inputs. Then, we use three real-world bug examples to illustrate the strong connections between bugs and execution paths.

4.1.1 *High Overhead of Sanitizers*

Despite the fact that sanitizers are highly effective in exposing software bugs, they are initially designed for software developers to conduct in-house testing rather than fuzzing. To benchmark sanitizer overhead in fuzzing, we use all 12 benchmark programs from our evaluation section. For each program, we compile five versions of it, *i.e.*, native program, ASan-enabled program, Debloat-enabled program, UBSan-enabled program, and MSan-enabled program. The native program refers to a normally built program without using any sanitizers. Since Debloat [165] achieves the state-of-the-art optimization for ASan, we include it to understand the significance of its improvement. We use AFL++ as the default fuzzer, and for each program:

Step (1) Use AFL++ to fuzz the native program and collect the first *one million* generated inputs to the program. All these inputs are saved into disk[5] for future utilization.

5 We use tmpfs [157] to reduce I/O overhead.

Table 4.1: Execution speed, *i.e.*, number of executions per second, of native programs and sanitizer-enabled programs. The column "Slowdown" refers to the ratio of the native speed to the sanitizer speed. It is calculated by dividing the native speed by the sanitizer speed. ✗ indicates a compilation failure.

Programs	**Native**	**ASan**		**Debloat**		**UBSan**		**MSan**	
	Speed	**Speed**	**Slowdown**	**Speed**	**Slowdown**	**Speed**	**Slowdown**	**Speed**	**Slowdown**
imginfo	2,964	869	3.4	907	3.3	1,968	1.5	43	68.4
infotocap	2,676	685	3.9	✗	✗	1,962	1.4	43	62.1
jhead	2,963	859	3.5	888	3.3	2,652	1.1	45	66.1
mp3gain	1,488	627	2.4	634	2.4	917	1.6	42	35.3
mp42aac	1,917	472	4.1	✗	✗	682	2.8	42	46.1
mujs	1,491	425	3.5	440	3.4	685	2.2	42	35.9
nm	2,209	586	3.8	✗	✗	1,597	1.4	43	50.8
objdump	573	212	2.7	✗	✗	250	2.3	38	15.1
pdftotext	410	151	2.7	✗	✗	192	2.1	35	11.7
tcpdump	1,754	432	4.1	493	3.6	561	3.1	42	42.0
tiffsplit	2,093	665	3.2	✗	✗	1,247	1.7	43	48.7
wav2swf	2,757	486	6.0	517	5.5	1,211	2.5	43	63.6
Average	1,941	539	3.6	-	-	1,160	2.0	42	45.5

Table 4.2: Ratio of bugger-triggering inputs.

Programs	**Executions**	**Bug-triggering**	**Ratio**
imginfo	8.4M	92,345	1.1%
infotocap	14.0M	23,784	0.2%
jhead	16.7M	468,202	2.8%
mp3gain	13.9M	105,349	0.8%
mp42aac	4.6M	16	<0.01%
mujs	13.5M	19,134	0.1%
nm	11.6M	517	0.0%
objdump	10.4M	328,546	3.2%
pdftotext	7.3M	1,358	<0.01%
tcpdump	8.6M	10,980	0.1%
tiffsplit	11.6M	23,017	0.2%
wav2swf	7.7M	563,588	7.3%
Average	10.7M	136,403	**1.3%**

Step (2) Run AFL++ again on the native program to benchmark its running time on the saved one million inputs. The AFL++ here is slightly modified to fetch inputs from the disk instead of generating them.

Step (3) Repeat **Step(2)** on four sanitizer-enabled programs to collect their running time on the same set of inputs.

We ran the above experiment 10 times and reported the average fuzzing speed. All experimental settings are the same as our later evaluation in Section 4.3.1. Table 4.1 presents the average speed, *i.e.*, number of executions per second, of each program. Compared to native programs, ASan, UBSan, and MSan averagely reduce the speed by 3.6x, 2.0x, and 45.5x, respectively. Specifically, ASan reduces the speed by 2.4x∼6.0x, UBSan by 1.1x∼3.1x, and MSan by 11.7x∼68.4x. Even for the best ASan optimization Debloat, its improvement over ASan is rather insignificant compared to the native program. Such huge sanitizer overheads inevitably hinder the fuzzing throughput. Because sanitizers bring fuzzing a significantly stronger bug-detection capability, current fuzzers have to bear the following speed loss.

Suppose that we have a way to reduce or even eliminate sanitizers' overhead while still keeping their bug-detection capability, fuzzing would then benefit significantly from it.

4.1.2 *Rareness of Bug-triggering Inputs*

Fuzzers typically generate a large body of inputs for a target program. It is intuitive that bug-triggering inputs are rarely met during fuzzing. To understand the ratio of bug-triggering inputs to all the generated inputs, we count the total number of bug-triggering inputs and all generated inputs during 24 hours of fuzzing. The experimental data is from our later evaluation in Section 4.3.3.

Table 4.2 lists the number of total generated inputs, the number of bug-triggering inputs, and the ratio. We can find that averagely *only 1.3%* of inputs are bug-triggering. For some programs, it is even rarer. For instance, on pdftotext, less than 1 out of 10^4 inputs trigger bugs. We can conclude that *Only a tiny fraction of fuzzer-generated inputs are bug-triggering.* Blindly sanitizing all of them is thus a huge waste of resources.

4.1.3 *Illustrative Examples*

In this section, we present examples of real-world bugs to demonstrate that inputs that trigger bugs follow distinct execution paths not seen with normal inputs, further reinforcing the rationale behind our approach.

Buffer-Overflow. A Buffer-Overflow bug is triggered when buffer access exceeds the allocated range of stack or heap memory. Figure 4.1 shows a real-world Buffer-Overflow bug from wav2swf in CVE-2017-11099. The buggy buffer access is located in line 8. When the two `for` loops iterate a significant number of times, the offset `pos2+j` exceeds the buffer range of `dest->data`. The original cause is from the large values of both `src->size` and `fill`. But these unusual data subsequently lead to a unique execution path, *i.e.*, a long chain of executions $\{1 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 8 \rightarrow 9 \rightarrow \cdots \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow \cdots\}$. In practice, we observe that data-sensitive bugs like Buffer-Overflow often accompany control-flow changes that normal executions do not exercise. Thus, using unique execution path as the indicator for sanitization can help us encapsulate bug-triggering inputs. The many Buffer-Overflow bugs identified by SAND in our evaluation will further confirm this rationale.

```
1  int wav_convert2mono(struct WAV *dest, int rate)
2  {
3      ...
4      for(i=0; i < src->size; i += channels) {
5        int j;
6        int pos2 = ((int)pos)*2;
7        for(j=0;j < fill; j += 2) {
8          dest->data[pos2+j+0] = 0;
9          dest->data[pos2+j+1] = src->data[i]+128;
10       }
11       pos += ratio;
12     }
13     ...
14 }
```

Figure 4.1: A simplified Buffer-Overflow bug from wav2swf in CVE-2017-11099. Line 8 triggers a buffer overflow when the two `for` loop iterations significantly change the buffer offset "`pos2+j`".

```
1  void JBIG2Stream::
2  readTextRegionSeg(Guint segNum, ...)
3  {
4      ...
5      numSyms = 0;
6      for (i = 0; i < nRefSegs; ++i) {
7        if ((seg = findSegment(refSegs[i]))) {
8          if (seg->getType()==jbig2SegSymbol) {
9             numSyms += seg->getSize();
10         } else if (seg->getType() == jbig2Se) {
11                codeTables->append(seg);
12         }
13     ...
14     syms = (JBIG2Bitmap **)gmallocn(numSyms);
15     ...
16 }
```

Figure 4.2: A simplified Integer-Overflow bug from xpdf (containing pdftotext) in CVE-2022-38171. Line 9 triggers an integer overflow in `numSyms` when the `if` branch in line 8 is evaluated to `True` many times.

Integer-Overflow. Figure 4.2 shows an Integer-Overflow bug in line 9, where the variable `numSyms` overflows its valid range when the `if` guard in line 8 is frequently evaluated to true. Although integer overflow is a locally data-sensitive bug type, it can be identified through unusual control-flow visits. In this example, the overflowed value in `numSyms` subsequently causes a small allocated buffer in line 14, which leads to buffer overflow and dramatic control-flow changes in the rest of the execution.

Use-of-Uninitialized-Memory. Figure 4.3 shows a Use-of-Uninitialized-Memory bug in line 24, where the `while` loop is conditioned on an uninitialized memory pointed to by `tmp`. The buffer `tmp` is obtained via the function call to `_nc_tic_expand()`, which returns an allocated buffer. Normally, `buffer` will be initialized either in line 9 or in lines 11 and 13. The corresponding (partial) execution paths are $\{7 \rightarrow 9 \rightarrow 18 \rightarrow 24\}$, $\{7 \rightarrow 11 \rightarrow 18 \rightarrow 24\}$, and $\{7 \rightarrow 13 \rightarrow 18 \rightarrow 24\}$. Buggy executions, however, jump from line 7 to line 15 without initializing `buffer`. The corresponding buggy execution path is $\{7 \rightarrow \mathbf{15} \rightarrow 18 \rightarrow 24\}$, which is different from all normal executions.

We have illustrated that both control-flow-related and data-sensitive bugs can result from/in unusual execution paths. This observation motivates us to utilize the execution path for determining whether the expensive sanitizer checks should be invoked.

4.2 APPROACH

This section introduces the design of our new fuzzing framework SAND. Section 4.2.1 defines *execution path* and its proxy approximation, *execution pattern*. Section 4.2.2 describes our proposed fuzzing framework. Section 4.2.3 clarifies the technical details of the implementation.

4.2.1 *Preliminary: Execution Path and its Proxy*

Our illustrative bugs exemplify that bug-triggering inputs have unique execution paths. We formally define the execution path as follows:

Definition 4 (Execution Path). *Given an execution $\mathcal{E}$, the* execution path *of $\mathcal{E}$ is defined as $\Pi_{\mathcal{E}} = [e_1, e_2, \cdots, e_n]$, where e_i is the unique id of the code edge executed by $\mathcal{E}$. Note that, $\Pi_{\mathcal{E}}$ is ordered meaning that e_i is executed before e_j if $i < j$.*

```
1  char *
2  _nc_tic_expand(const char *src, bool tic_format)
3  {
4      static char *buffer;
5      ...
6      buffer = typeRealloc(char, length, buffer);
7      int bufp = 0;
8      if (ch == '%' && REALPRINT(str + 1))
9         buffer[bufp++] = *str++;
10     else if (ch == 128)
11        buffer[bufp++] = '\\';
12     } else if (ch == '\033')
13        buffer[bufp++] = '\\';
14     else
15         bufp += 4;
16     str++;
17     ...
18     return buffer;
19 }
20
21 int fmt_entry(...) {
22     ...
23     char *tmp = _nc_tic_expand(boxchars, 1);
24     while (*tmp != '\0') {
25         ...
26     }
27 }
```

Figure 4.3: A simplified Use-of-Uninitialized-Memory bug from infotocap in CVE-2019-17595. The buffer `*tmp` in line 24 is uninitialized when the execution jumps from line 7 to line 15.

Execution path is a temporal transition sequence of all executed code when executing an input on the target program. It contains the full information on control-flow visits. For instance, the buggy execution in Figure 4.3 $\{7 \rightarrow 15 \rightarrow 18 \rightarrow 24\}$ has the execution path as $[7, 15, 18, 24]$. Execution path is order-sensitive meaning that $[7, 15, 18, 24] \neq [15, 7, 18, 24]$. Unfortunately, obtaining the *execution path* of execution is too expensive to be practical in fuzzing [40, 146]. Since throughput is a key factor in fuzzing effectiveness, we cannot directly use *execution path* in fuzzer design. In this thesis, we propose to use *execution pattern* as an approximate yet accurate proxy for the execution path. We define *execution pattern* as follows:

Definition 5 (Execution Pattern). *Given an execution $\mathcal{E}$, the* execution pattern *of $\mathcal{E}$ is defined as $\mathcal{T}_{\mathcal{E}} = \{e_1, e_2, \cdots, e_m\}$, where $e_i \neq e_j (i \neq j)$ and e_i is the unique id of the code edge reached by $\mathcal{E}$. Note that, $\mathcal{T}_{\mathcal{E}}$ is order-insensitive,* e.g., *$\{e_1, e_2, e_3\} = \{e_2, e_3, e_1\}$.*

Execution pattern records all executed code edges of an execution. For instance, the buggy execution in Figure 4.3 $\{7 \rightarrow 15 \rightarrow 18 \rightarrow 24\}$ has the execution pattern as $\{7, 15, 18, 24\}$. Execution pattern is order-insensitive, *e.g.*, $\{7, 15, 18, 24\} = \{15, 7, 18, 24\}$. Intuitively, the execution pattern inevitably loses precision in approximating the execution path. But it still captures the uniqueness of buggy executions. For the bug in Figure 4.3, normal execution patterns are $\{7, 9, 18, 24\}$, $\{7, 11, 18, 24\}$, and $\{7, 13, 18, 24\}$. The buggy execution pattern $\{7, 15, 18, 24\}$, although containing no ordering information, is distinct from normal ones. Our evaluation in Section 4.3.2 will use extensive experiments to demonstrate that execution patterns can precisely encapsulate buggy executions.

An essential benefit of the execution pattern is its ease of acquisition during fuzzing. Fuzzers like AFL++, by default, utilize an efficient data structure, bitmap, to collect visited code edges of an execution. Figure 4.4 shows an example of this procedure. The bitmap is initialized to all zeros for a new execution. For the execution path $[5, 3, 9, 7]$, the corresponding positions in the bitmap are marked. This bitmap is then used to update a global coverage map with a logic OR. For the next execution $[1, 7, 9, 2]$, a similar bitmap is initialized and then marked. The coverage map is then cumulatively updated to record all code edges visited by all previous executions. The design of *execution pattern* allows us to obtain it effortlessly from the bitmap of execution, as shown in the middle of Figure 4.4. Although the execution pattern might lose precision in modeling the execution path, we view it as a trade-off for performance. Furthermore, our evaluation will

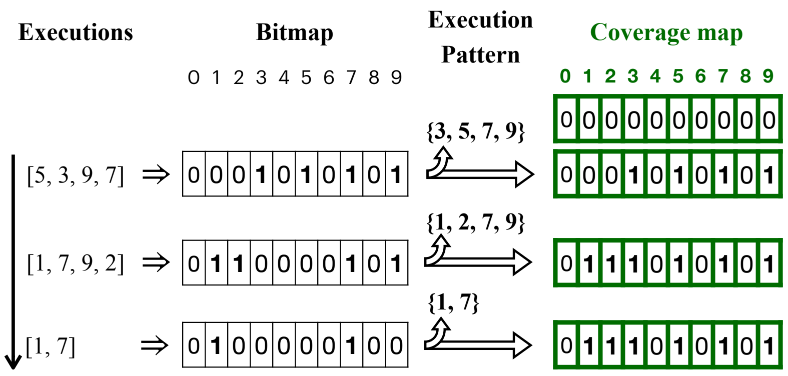


Figure 4.4: Executions (left) are monitored through bitmaps (middle), which are used in AFL++ to update the coverage map (right). Our execution patterns (bottom right) can be derived from these bitmaps.

highlight that *execution pattern* is accurate enough in encapsulating buggy executions.

4.2.2 *Sanitization-decoupled Fuzzing*

Based on the above formalization to execution patterns, we introduce our new fuzzing framework design. We first describe the general workflow of a coverage-guided fuzzer (CGF). The gray area in Figure 4.5 outlines the high-level fuzzing sketch of a CGF. *Before fuzzing starts, the fuzzer compiles the target program with fuzzer instrumentation and/or sanitizers.* Then, it fuzzes the target as follows:

(1) **Seed selection.** Select one seed from the seed pool according to predefined strategies.

(2) **Mutation.** Mutate the seed to generate new test inputs.

(3) **Executing on the target program.** Execute a test input on the target program.

(4) **Coverage and execution analysis.** Collect coverage feedback from the execution. If the execution increases coverage, save it to the seed pool; if the execution results in a crash, report the corresponding input as bug-triggering and save it to the disk; otherwise, discard it.

As one can see, CGFs rely on the execution result of the target program to detect bugs. In order to maximize bug detection capability, current

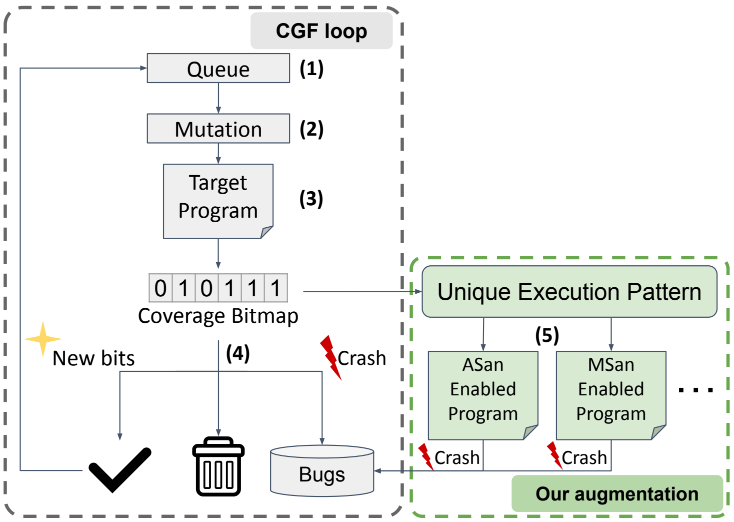


Figure 4.5: SAND fuzzing loop.

CGFs usually compile the target program with sanitizers enabled. This routine significantly slows down fuzzing speed due to the high overhead of sanitizers.

In this thesis, we tackle this problem by decoupling sanitization from the conventional fuzzing loop. The green part in Figure 4.5 highlights our approach. Before fuzzing starts, the fuzzer compiles multiple versions of the same program: (1) a normally built program without any sanitizer enabled (denoted as $\mathcal{P}_{fuzz}$), *on which the fuzzer performs fuzzing*, and (2) a set of sanitizer-enabled programs, *e.g.*, ASan-enabled program ($\mathcal{P}_{ASan}$) and MSan-enabled program ($\mathcal{P}_{MSan}$). The fuzzer follows the same steps as a CGF to fuzz the *normally built program*. But, after each execution of the target program, we introduce a new step:

(5) **Conditional sanitization.** Extract the *execution pattern* of the execution from its bitmap. If the execution pattern has been observed before, *i.e.*, not unique, discard it. Otherwise, the current input is identified as *sanitization-required*. The fuzzer executes this input on each sanitizer-enabled program ($\mathcal{P}_{ASan}$, $\mathcal{P}_{MSan}$, *etc.*) and reports any discovered crashes.

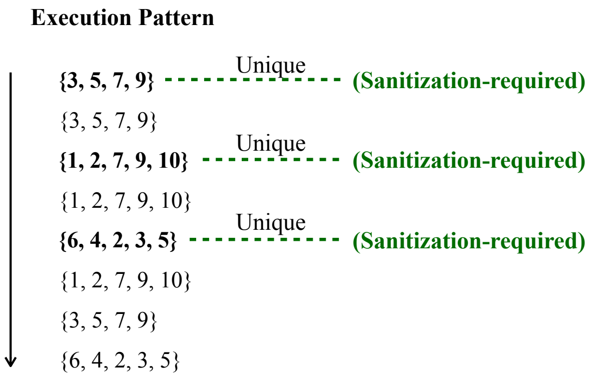


Figure 4.6: Out of all eight consecutive executions from top to bottom, three are identified as unique and require sanitization.

Example. Figure 4.6 illustrates the process of identifying sanitization-required inputs. Starting from the first execution with pattern $\{3, 5, 7, 9\}$, the fuzzer identifies it as a unique execution pattern and thus sanitization-required. The second execution has the same pattern as before; thus, sanitization is unnecessary. Similarly, the third and fifth executions have unique patterns not seen before and thus require sanitization. Our hypothesis is that all input triggering unique bugs also have unique execution patterns. Assuming that the fifth execution $\{6, 4, 2, 3, 5\}$ is buggy, the fuzzer can successfully identify the bug during sanitization. ***Since executions holding the same execution path are likely to have similar semantics,* e.g.*, exercising the same functionality or triggering the same bug,*** we only need to sanitize one buggy execution from the same set of unique execution paths to identify the bug. In this example, the first time we sanitize the execution with pattern $\{6, 4, 2, 3, 5\}$, we can discover the bug. For all future executions with the same pattern, they are likely to trigger the same bug and do not need sanitization. Our evaluation in Section 4.3.3 will support this claim.

This newly introduced **conditional sanitization** does not alter the standard fuzzing logic. The execution pattern is obtained from the already-available bitmap collected on the normally built target program. To determine whether or not an execution pattern has been observed before, we use a hash table to store all observed execution patterns (see details in Section 4.2.3).

Algorithm 5 sketches the implementation pseudo-code of our new fuzzing framework. In each fuzzing loop (line 1), the fuzzer first selects a seed s

Algorithm 5: The New Fuzzing Loop of SAND

```
Input: Seed pool $\mathcal{S}$.
1  while ¬Abort() do
2      $s \leftarrow$ SelectSeed($\mathcal{S}$) // Seed selection
3      $s' \leftarrow$ Mutate($s$) // Generate input
4      $ret, bitmap \leftarrow$ Execute($s', \mathcal{P}_{fuzz}$)
5      $\mathcal{T}_{\mathcal{E}} \leftarrow$ GetExecutionPattern($bitmap$)
6      if IsUnique ($\mathcal{T}_{\mathcal{E}}$) then
7          foreach $\mathcal{P}_{san} \in \{\mathcal{P}_{ASan}, \mathcal{P}_{MSan}, \cdots\}$ do
8              $ret_{san} \leftarrow$ SanExecute($s', \mathcal{P}_{san}$)
9              if $ret_{san} == crash$ then
10                 $ret = crash$
                                                   // Our augmentation
11     if $ret == crash$ then // Crash?
12         save $s'$ to disk
13     if covers new code then // New coverage?
14         add $s'$ to $\mathcal{S}$
```

and mutates it to generate a new input s' (lines 2-3). Next, it executes the normally built program $\mathcal{P}_{fuzz}$ on the input to collect its execution return *ret* and bitmap *bitmap* (line 4). Then, the fuzzer extracts the execution pattern $\mathcal{T}_{\mathcal{E}}$ from *bitmap* (line 5) and determines whether or not this execution pattern has been observed (line 6). If $\mathcal{T}_{\mathcal{E}}$ is new, the fuzzer labels it as sanitization-required and executes each of the available sanitizer-enabled programs $\mathcal{P}_{san}$ on the input s' (lines 7-8). Meanwhile, $\mathcal{T}_{\mathcal{E}}$ will be added to the hash table. If any execution crashes, meaning the input s' triggers a bug, the fuzzer sets the return status to *crash* (lines 10-11). Finally, the fuzzer continues the original procedure: save the new input as bug-triggering if the execution return status is *crash* (lines 11-12); or queue it to the seed pool if it increases coverage (lines 13-14).

Our new fuzzing framework decouples sanitization from standard fuzzing logic. It has the following main advantages:

- ***Orthogonal to CGFs.*** We only introduce an orthogonal step to execute sanitizer-enabled programs on inputs with unique execution patterns. In theory, any AFL-family fuzzers can be augmented by our approach without modifying their main fuzzing logic.

Algorithm 6: Identify unique execution patterns

```
1 IsUnique($\mathcal{T}_{\mathcal{E}}$):
2     $cksum \leftarrow Hash(\mathcal{T}_{\mathcal{E}})$
3     if HashTable[$cksum$] $\neq 1$ then
4         HashTable[$cksum$] $= 1$
5         return $True$;
6     return $False$;
```

- ***Sanitizer inclusive.*** Normally, some sanitizers like ASan and MSan are mutually exclusive, meaning that they cannot be enabled on the same program. Current fuzzers can only perform fuzzing on a program with only one of such sanitizers enabled. In our new framework, multiple sanitizer-enabled programs can be used for sanitization simultaneously. We will provide additional technical details in Section 4.2.3 to explain how we support multiple sanitizers.

- ***Effective.*** Our evaluation will show that only a small fraction (averagely $\leq 2\%$) of inputs have unique execution patterns and require sanitization, thus significantly improving fuzzing throughput.

- ***Practical.*** First, sanitizers are directly used for instrumentation, and thus, we do not need to change their code base. Second, the only modification we applied to a fuzzer is the augmented new step after each execution. Other parts of the fuzzer are not touched.

4.2.3 *Implementation*

Unique Execution Pattern Analysis. We obtain the execution pattern of an execution from its bitmap. In our implementation, we use the function `simplify_trace()` in AFL++ to achieve this goal. This design and implementation allow us to efficiently get execution patterns during fuzzing. To identify unique execution patterns, we calculate checksums of all observed execution patterns and use a hash map to store them. Algorithm 6 details the pseudocode of the process. The hash table HashTable is initialized to all zeros at the start of fuzzing. In our implementation, we use XXH32 hashing algorithm[158] because of its fast speed. The size of HashTable is set to 32-bit, which supports a maximum of $4,294M$ different checksums. Our evaluation in Section 4.3.6 demonstrates that the cost of hashing is negligible, and no instances of hash collision are observed.

Note that normal executions can also have unique execution patterns, such as the first and third executions in Figure 4.6. Our approach is efficient as long as the overall ratio of unique execution patterns during fuzzing is low. As our evaluation in Section 4.3.4 shows, the average ratio is $< 2\%$.

Program Instrumentation in Sand. The fuzz target $\mathcal{P}_{fuzz}$ is instrumented by SAND to include the necessary instrumentation code for coverage collection. Since all the sanitizer-enabled programs are used for sanitization only, no such instrumentation is needed. Thus, we directly use the LLVM compiler to compile the program with different sanitizers. Because ASan and MSan are mutually exclusive, we combine them in SAND. By default, we use two sanitizer-enabled programs, *i.e.*, ASan/UBSan-enabled program ($\mathcal{P}_{ASan/UBSan}$) and MSan-enabled program ($\mathcal{P}_{MSan}$). To reduce the burden of invoking $\mathcal{P}_{ASan/UBSan}$ and $\mathcal{P}_{MSan}$, we utilize the *forkserver*[161] mode to create one forkserver to communicate with all sanitizer-enabled programs efficiently during fuzzing.

4.3 EVALUATION

We implemented SAND based on AFL++-4.05c [34], the latest version at the time of implementation. AFL++ is the state-of-the-art grey-box fuzzer and has been widely used as the baseline fuzzer in many previous study [12, 84, 100]. Our evaluation first evaluates the accuracy of execution pattern (Section 4.3.2) in encapsulating buggy executions and then extensively evaluates the end-to-end fuzzing performance of SAND in terms of bug-finding (Section 4.3.3), throughput (Section 4.3.4), and code coverage (Section 4.3.5).

4.3.1 *Experimental Setup*

Benchmark. We use real-world programs from the benchmarking test platform UNIFUZZ for our evaluation. We use the provided seeds from UNIFUZZ for all fuzzing campaigns. To maximally understand SAND's capability in different sanitizers, we use all three popular sanitizers, *i.e.*,

Table 4.3: All 12 real-world programs from UniFuzzused in the evaluation.

Type	Program	Type	Program
Image	imginfo	Text	infotocap
	jhead		mujs
	tiffsplit		pdftotext
Audio	mp3gain	Binary	nm
	wav2swf		objdump
Video	mp42aac	Network	tcpdump

ASan, UBSan, and MSan. Due to the compatibility issue of MSan, we failed to instrument 8 out of 20 programs with MSan. We thus exclude them from our evaluation. Our full evaluation is done on the remaining 12 programs. These programs Table 4.3 lists the details. These programs cover a diverse range of input types, including image (*e.g.*, imginfo), audio (*e.g.*, wav2swf), video (*e.g.*, mp42aac), text (*e.g.*, infotocap), binary (*e.g.*, objdump), and network packet (*e.g.*, tcpdump).

Baseline. Since ASan and UBSan are compatible with each other, we combine them together when building binaries. For each program fuzzed in Sand, we compile a normally built binary $\mathcal{P}_{fuzz}$ and two sanitizer-enabled binaries, *i.e.*, $\mathcal{P}_{ASan/UBSan}$ and $\mathcal{P}_{MSan}$.

We choose AFL++ as the baseline fuzzer and use it to fuzz (1) normally built programs (denoted as "AFL++-Native"), (2) ASan/UBSan-enabled programs (denoted as "AFL++-ASan/UBSan"), and (3) MSan-enabled programs (denoted as "AFL++-MSan"). All fuzzers and programs are built with LLVM-14, the latest stable version at the time of implementation.

To understand if Sand can surpass the existing sanitizer optimization schemes, we also choose the state-of-the-art ASan optimization technique, Debloat [165]. Because Debloat optimizes ASan, it can also be used together with UBSan. To maximize its bug detection capability, we let AFL++ to fuzz on Debloat/UBSan-enabled program (denoted as "AFL++-Debloat/UBSan"). All programs instrumented with Debloat are built with LLVM-12 because this is the highest LLVM version that Debloat supports. Since compiling infotocap, mp42aac, nm, objdump, and pdftotext

Table 4.4: Ratios of inputs that have unique execution patterns. “All” refers to the ratio of all generated inputs that have unique execution patterns. “Bug” refers to the ratio of bug-triggering inputs that have unique execution patterns.Ratios of inputs that have unique execution patterns. “All” refers to the ratio of all generated inputs that have unique execution patterns. “Bug” refers to the ratio of bug-triggering inputs that have unique execution patterns.

Programs	All	Bug
imginfo	0.22%	72.8%
infotocap	8.11%	98.3%
jhead	1.44%	91.6%
mp3gain	0.18%	98.2%
mp42aac	0.04%	100.0%
mujs	5.34%	99.0%
nm	0.35%	100.0%
objdump	1.30%	100.0%
pdftotext	4.82%	100.0%
tcpdump	2.09%	100.0%
tiffsplit	1.17%	99.2%
wav2swf	0.01%	100.0%
Average	**2.09%**	**96.58%**

with Debloat results in compilation failures, we exclude them for AFL++-Debloat/UBSan.

Hardware and Setup. We conduct all experiments on a machine equipped with an AMD 3990x CPU and 256G memory running Ubuntu 22.04. Following Klee’s [65] standard we repeated all experiments 10 times and ran all fuzzing campaigns for 24 hours. We apply the Mann-Whitney U-test [103] to our results to understand their statistical significance.

4.3.2 *Effectiveness of Execution Pattern*

To understand if unique execution patterns can accurately encapsulate bug-triggering inputs/executions, we extensively analyze all executions during fuzzing. Specifically, we conduct the following experiments on the 12 programs:

Step (1) Use AFL++ to fuzz the *normally built* program.

Step (2) For each generated input, we first obtain its execution pattern, then examine whether or not this execution pattern is unique, *i.e.*, has been observed before.

Step (3) For each input, no matter whether or not its execution pattern is unique, we run it on ASan- and UBSan-enabled programs to test if it triggers a bug.

We ran the experiments for 24 hours and repeated them ten times. Because we need to run through sanitizer-enabled programs for every input in order to gather a sufficiently large number of inputs, we exclude MSan due to its deficient speed. With this set of experiments, we want to answer **Q1:** out of all executions in **Step (2)**, how many of them are marked as having unique execution patterns? **Q2:** out of all bug-triggering inputs/executions in **Step (3)**, how many of them are also marked as having unique execution patterns in **Step (2)**?

The first question **Q1** can tell us the ratio of unique execution patterns during fuzzing. A smaller ratio indicates that sanitization is required less frequently, resulting in higher speed. The second question **Q2** can inform us how effective the unique execution pattern is in encapsulating bug-triggering inputs. Table 4.4 shows the result. The second column shows that, on average, only 2% inputs have unique execution patterns. For more than half of the programs, the ratio is even below 1%. We can thus conclude that *Only a small fraction of fuzzer-generated inputs have unique execution patterns.* The third column shows that more than 96% of bug-triggering inputs have unique execution patterns. This means that if we only pass inputs with unique execution patterns to sanitizer-enabled programs, we can successfully sanitize 96% of the buggy inputs. Note that we do not deduplicate all bug-triggering inputs here; most of them are, in fact, duplicates. Considering the same bug can be triggered multiple times during fuzzing, 96% accuracy can already ensure, with a high probability, that no bugs will be missed in practice. Our upcoming evaluation will provide extensive end-to-end evaluation results to confirm this high precision.

4.3.3 *Bug-Finding Capability*

Finding bugs is the ultimate goal of fuzzing. In this section, we evaluate the bug-finding capability of all fuzzers. In particular, we would like to answer the following two questions:

Q1 Does SAND find *more bugs* compared to other fuzzers?

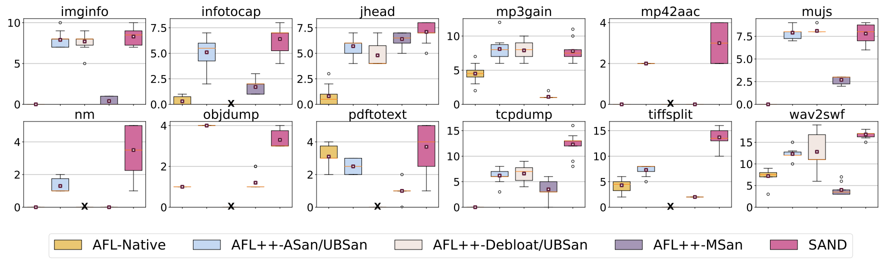


Figure 4.7: The number of unique bugs detected by fuzzers across repetitions. ✗indicates a compilation failure.

Table 4.5: The mean number of unique bugs across repetitions. The ✗ indicates a compilation failure. The largest mean numbers are highlighted in green .

Programs	**Native**	**AFL++- ASan/ UBSan**	**AFL++- Debloat/ UBSan**	**AFL++- MSan**	**Sand**	p-val
imginfo	0	7.9	7.7	0.4	8.3	0.18
infotocap	0.3	5.1	✗	1.7	6.4	0.03
jhead	0.8	5.7	4.8	6.4	7.1	0.03
mp3gain	4.5	8.1	7.9	1.1	7.8	0.71
mp42aac	0	2	✗	0	3	0.00
mujs	0	7.9	8.1	2.7	7.8	0.75
nm	0	1.3	✗	0	3.5	0.00
objdump	1	4	✗	1.2	3.3	1.00
pdftotext	3.1	2.5	✗	1	3.7	0.09
tcpdump	0	6.2	6.6	3.5	12.3	0.00
tiffsplit	4.3	7.3	✗	2	13.7	0.00
wav2swf	7.2	12.3	12.8	4	16.8	0.03
Average	1.8	5.9	-	2.0	**7.8**	

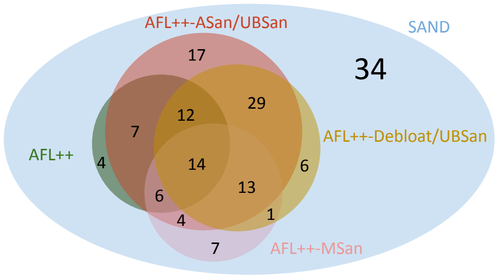


Figure 4.8: Unique bugs found by fuzzers.

Q2 Does SAND *miss any bugs* found by other fuzzers?

To answer these questions, we collect all crashes found by each fuzzer. We triage all crashes according to their root causes to quantify the number of unique bugs each fuzzer finds. Our deduplication is done with both the stack frame information from GDB [80] and manual analysis.

Number of Unique Bugs. We plot the number of unique bugs in every repetition in Figure 4.7. Table 4.5 aggregates the results and reports the mean number of unique bugs. On average, SAND finds 32% more (7.8) bugs than the second-best fuzzer (5.9 in AFL++-ASan/UBSan). On 7 out of 12 programs, SAND found more bugs than all other fuzzers with statistical significance (*i.e.*, p-value < 0.05). On some programs, SAND can cover significantly more (> 50%) bugs. For instance, on tcpdump, SAND can averagely find 12.3 bugs, while the next-best fuzzer AFL++-Debloat/UBSan only finds 6.6 bugs. For the remaining five programs, SAND finds *statistically the same* (p-value > 0.05) number of unique bugs. In summary, our result answers **Q1**: SAND *has a significantly stronger bug-finding capability than all other fuzzers.*

Compared to AFL++-Native, ASan, as well as Debloat and UBSan, has positive effects on 11 out of 12 programs. MSan finds fewer bugs on four programs, which is due to its extremely low fuzzing speed. An interesting case is pdftotext, where AFL++-Native can detect more bugs than other fuzzers except for SAND. The main reason is that all bugs in pdftotext can be triggered without sanitizers, while AFL++-Native has higher throughput than all other fuzzers. Nevertheless, the overall result confirms the necessity of using sanitizers during fuzzing.

Table 4.6: Accumulative number of unique bugs. ✗indicates a compilation failure. "All" refers to the number of unique bugs found together by all other fuzzers except for SAND.

Programs	**AFL++-**				**All**	**Sand**
	Native	**ASan/ UBSan**	**Debloat/ UBSan**	**MSan**		
imginfo	0	10	10	1	11	11
infotocap	1	8	✗	3	9	10
jhead	5	9	8	10	10	12
mp3gain	10	12	12	2	12	14
mp42aac	0	2	✗	0	2	4
mujs	0	9	9	3	9	10
nm	0	2	✗	0	2	7
objdump	1	4	✗	2	4	4
pdftotext	5	3	✗	3	5	5
tcpdump	0	12	15	9	23	36
tiffsplit	8	9	✗	2	11	18
wav2swf	13	22	21	10	22	23
Sum	43	102	-	45	120	**154**

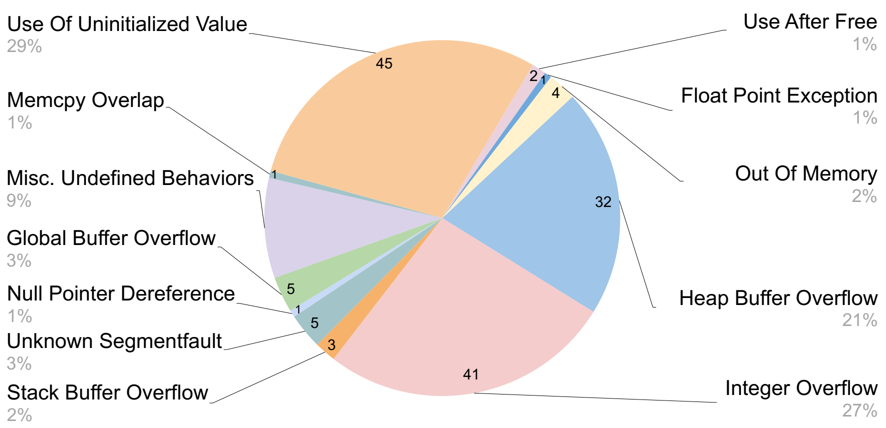


Figure 4.9: The distribution of diverse types of bugs found by SAND.

Accumulative Number of Unique Bugs. To understand the overlaps of bugs found by different fuzzers, we accumulate all unique bugs found by each fuzzer in each repetition. Figure 4.8 illustrates the unique bug sets of different fuzzers through a Venn diagram. It shows that SAND *covers all bugs* discovered by all other fuzzers. Moreover, SAND finds **34** additional bugs that all other fuzzers can not cover. Table 4.6 breaks down the total number of bugs discovered by each fuzzer. Compared to other fuzzers, SAND finds more bugs on all programs. The second-to-last column "All" lists the total number of bugs found by *all other fuzzers together*. Even compared to "All", SAND can still find more bugs in each program. On some programs, SAND can even cover 2x $\sim$ 3x more bugs. For instance, SAND discovers 2x more bugs on mp42aac and 3.5x more bugs on nm. In summary, we can answer **Q1** and **Q2**: SAND *does not miss any bugs and can find significantly more bugs.*

Number of Unique Bugs Reported by Sanitizer-enabled Programs. Of all the 154 unique bugs identified by SAND, more than 75% of them are not detectable on the normally built programs. These bugs are reported after invoking sanitizer-enabled programs in SAND.

Bug Types. Our approach relies on the control-flow information to identify sanitizer-required inputs. Our observation, as illustrated with bug examples in Section 4.1.3, is that data-sensitive bugs like buffer overflow and integer overflow usually result from/in control-flow changes. Figure 4.9 reports

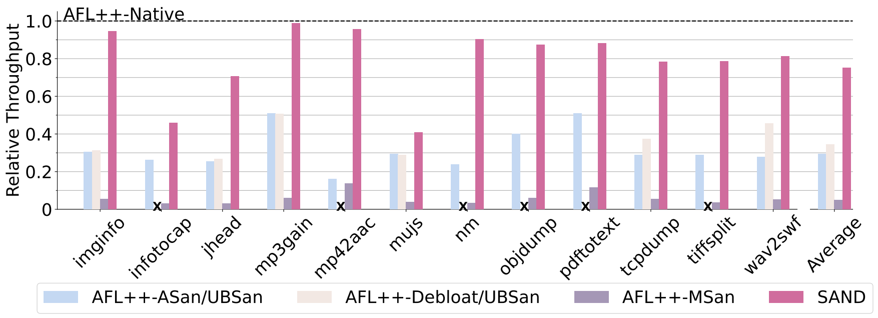


Figure 4.10: Relative throughput normalized to AFL++-Native. ✗indicates a compilation failure. "Average" refers to the average throughput of all programs.

the types of bugs found by Sand. Firstly, Sand can indeed find many control-flow-related bugs, such as the use of uninitialized memory (45 bugs) and use after free (2 bugs). The small number of use-after-free bugs is because they are indeed relatively rare in practice [80]. Secondly, Sand can find a large number of data-sensitive bugs, such as heap buffer overflow (32 bugs) and integer overflow (41 bugs). This outstanding result confirms our assumption that data-sensitive bugs can be captured by control-flow information. Given that Sand does not miss any bugs and finds both control-flow and data-sensitive bugs, we can conclude that *our approach generally applies to all kinds of sanitizer-detectable bugs.*

False Negatives. Despite the outstanding performance of Sand, we have no guarantee that Sand can cover all bugs. Theoretically, Sand may have false negatives where certain bugs are missed. This false negative impact can be inferred from the mujs performance in Table 4.5, where Sand has a slightly lower mean number of bugs (7.8) than AFL++-ASan/UBSan (7.9) and AFL++-Debloat/UBSan (8.1). The reason is that one bug in mujs was not triggered in all repetitions of Sand, but in all repetitions in the other two fuzzers. The execution pattern for this bug can sometimes be seen in normal executions, which leads to the less frequent discovery in Sand. Due to the stochastic nature of fuzzing, triggering a bug only once is sufficient for Sand to detect it in practice. Given the overall superior performance of Sand, we believe that the moderate false negative issue is acceptable and does not impede its general effectiveness.

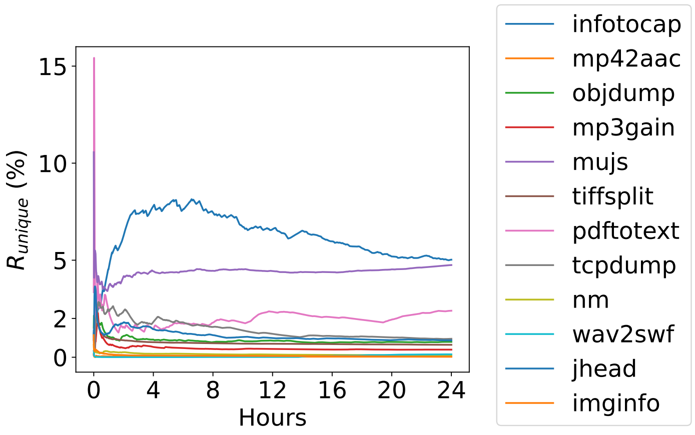


Figure 4.11: The trend of R_{unqiue}, *i.e.*, the unique execution pattern ratio, over time during fuzzing.

4.3.4 *Fuzzing Throughput*

We now analyze the end-to-end fuzzing throughput, *i.e.*, the total number of inputs generated and executed during fuzzing. Figure 4.10 shows the average throughput of each fuzzer normalized to AFL++-Native.

Compared to Sanitizers-enabled Fuzzers. SAND achieves an average of 2.6x, 2.1x, and 150x throughput than AFL++-ASan/UBSan, AFL++-Debloat/UBSan, and AFL++-MSan, respectively. Moreover, SAND has significantly higher throughput *on all programs*. On some of the programs, the speedup rate is even higher. For instance, on nm, SAND executes 4x and 303x more inputs than AFL++-ASan/UBSan and AFL++–MSan, respectively. *It is worth mentioning that* SAND *is equipped with all three sanitizers, including the slowest MSan. All other fuzzers, on the other hand, only support one or two sanitizers.*

Compared to AFL++-Native. Overall, SAND achieves 75% of AFL++-Native's throughput. On 4 out of 12 programs, SAND achieves more than 90% of AFL++-Native's throughput. *The result shows that* SAND *successfully increases the speed of fuzzing on sanitizer-enabled programs to a near-native level.*

Unique Execution Pattern Ratio. Intuitively, a smaller unique execution pattern ratio means that sanitizer-enabled programs are less frequently invoked and, consequently, have a higher throughput. To understand how this ratio changes during fuzzing, we track the unique execution pattern ratio every 40 seconds. Figure 4.11 plots the results. As expected, at the start of fuzzing, the ratio is relatively high because many execution patterns are new. With the fuzzing going on, more and more inputs have duplicate execution patterns, and thus, the ratio becomes significantly smaller. This tendency is analogous to the saturation situation of code coverage. Overall, SAND has less than 2% ratio on 10 out of 12 programs, meaning that only less than 2% of inputs are fed into sanitizer-enabled programs for sanitization. Although a slightly higher ratio is observed on infotocap and mujs, these ratios are eventually all less than 5%. According to the throughput data in Figure 4.10, SAND has indeed relatively lower speedup rates on these two programs.

4.3.5 *Code Coverage*

We use the `afl-showmap` utility in AFL++ to collect the code coverage. Table 4.7 presents the average code coverage achieved by each fuzzer on each program.

Compared to Sanitizers-enabled Fuzzers. SAND achieves statistically higher code coverage on 9 out of 12 programs. For the other three programs, SAND has higher code coverage but with no statistical significance. Intuitively, since SAND has a much higher throughput than all other sanitizer-enabled fuzzers, SAND executes more inputs and thus achieves higher code coverage.

Compared to AFL++-Native. On 6 out of 12 programs, there is no significant coverage difference between AFL++-Native and SAND. For the remaining six programs, SAND achieves almost the same code coverage as AFL++-Native, with an average of 0.53% less coverage. As analyzed before, SAND can achieve 75% throughput of AFL++-Native, which accounts for the coverage drop in some programs. Since bug-finding capability is the golden measuring metric for fuzzing, although AFL++-Native can achieve relatively higher code coverage, it has the worst bug-finding rate and thus is less favorable in practice.

Table 4.7: Code coverage (%) of fuzzers. ✗ indicates a compilation failure. "**Diff**" is the difference compared to AFL++-Native. The highest code coverage compared to AFL++-Native is highlighted in green.

Programs	AFL++ -Native	AFL++-						Sand	
		ASan/UBSan		Debloat/UBSan		MSan			
		Cov$_{p\text{-val}}$	Diff	Cov$_{p\text{-val}}$	Diff	Cov$_{p\text{-val}}$	Diff	Cov$_{p\text{-val}}$	Diff
imginfo	13.36	11.89 $_{0.00}$	-1.47	11.36 $_{0.00}$	-2.00	8.88 $_{0.00}$	-4.48	12.94 $_{0.04}$	-0.42
infotocap	19.42%	17.85 $_{0.03}$	-1.57	✗	✗	14.05 $_{0.03}$	-5.37	18.74 $_{0.27}$	-0.68
jhead	14.96	14.94 $_{0.37}$	-0.02	14.90 $_{0.00}$	-0.06	14.85 $_{0.37}$	-0.11	14.96 $_{1.00}$	0.00
mp3gain	41.57	38.52 $_{0.00}$	-3.05	38.60 $_{0.00}$	-2.97	34.47 $_{0.00}$	-7.10	39.88 $_{0.00}$	-1.69
mp42aac	7.15	6.80 $_{0.00}$	-0.35	✗	✗	6.78 $_{0.00}$	-0.37	7.04 $_{0.16}$	-0.11
mujs	27.97	21.04 $_{0.00}$	-6.93	20.79 $_{0.00}$	-7.18	21.18 $_{0.00}$	-6.79	27.26 $_{0.00}$	-0.71
nm	7.80	7.27 $_{0.00}$	-0.53	✗	✗	6.69 $_{0.00}$	-1.11	7.53 $_{0.01}$	-0.27
objdump	6.98	5.10 $_{0.00}$	-1.88	✗	✗	6.60 $_{0.00}$	-0.38	6.67 $_{0.00}$	-0.31
pdftotext	16.69	13.07 $_{0.00}$	-3.62	✗	✗	14.77 $_{0.00}$	-1.92	15.18 $_{0.00}$	-1.51
tcpdump	18.36	16.77 $_{0.21}$	-1.59	17.25 $_{0.38}$	-1.11	12.20 $_{0.21}$	-6.16	17.33 $_{0.47}$	-1.03
tiffsplit	20.96	17.93 $_{0.00}$	-3.03	✗	✗	13.34 $_{0.00}$	-7.62	20.59 $_{0.47}$	-0.37
wav2swf	2.04	1.96 $_{0.00}$	-0.08	1.89 $_{0.00}$	-0.15	1.60 $_{0.00}$	-0.44	2.00 $_{0.37}$	-0.04

Table 4.8: The hash overhead and collision in SAND.

Programs	NoHash	Sand-Hash		
	Speed	Speed$_{p\text{-val}}$	Overhead	Collision
imginfo	3,114	3,100 0.65	0.47%	0
infotocap	3,011	2,982 0.15	0.96%	0
jhead	3,327	3,327 0.94	0.00%	0
mp3gain	1,929	1,915 0.43	0.73%	0
mp42aac	1,702	1,688 0.21	0.87%	0
mujs	1,841	1,812 0.01	0.58%	0
nm	2,421	2,389 0.26	0.36%	0
objdump	1,279	1,266 0.36	0.05%	0
pdftotext	408	405 0.00	0.66%	0
tcpdump	2,018	2,004 0.52	0.73%	0
tiffsplit	2,335	2,334 0.94	0.02%	0
wav2swf	2,830	2,817 0.36	0.48%	0
Average	2,185	2,170	**0.74%**	**0**

4.3.6 *Hash in* SAND

As the Algorithm 5 indicates, SAND utilizes a hash table to store hash checksums for each execution pattern. In this section, we evaluate the hash overhead of SAND and the potential hash collision risk in the hash table.

Hash Overhead. We modified SAND to two versions to precisely evaluate the hash overhead. First, NOHASH, where lines 6-11 in Algorithm 5 are removed so that no hash operations are performed. Second, SAND-HASH, where lines 8-11 in Algorithm 5 are removed so that hash operations are performed as the normal SAND, but no sanitizer-enabled programs are invoked. We use the same random seed for both modified fuzzers to generate an identical set of inputs on the same initial seed pool. We run both fuzzers on each program ten times and record the total fuzzing time on the first *one million* inputs. The second and third columns in Table 4.8 report the average speed of both fuzzers. On 10 out of 12 programs, both fuzzers do not have statistically differentiable (p-val < 0.05) speed. Only on mujs and pdftotext, SAND-HASH is slightly slower at a rate of 1.6% and

0.7% while averagely SAND-HASH only incurs 0.7% penalty. We can then conclude that *hashing operations in* SAND *have negligible overhead to fuzzing*.

Hash Collision. *Hash collision* can happen when two different execution patterns either have the same hash checksum or result in the same index in the hash table. Because the effectiveness of SAND relies on the accurate identification of unique execution patterns, hash collisions may potentially harm the performance. To evaluate the hash collision rate of SAND, we use the SAND-HASH and expand it to save all execution patterns (rather than checksums) to disk. When an execution pattern is marked as observed, we compare this execution pattern with the saved execution pattern byte to byte. Any difference in the comparison signifies a hash collision. The last column in Table 4.8 lists the number of hash collisions for the first *one million* inputs. The result shows that *none hash collision* was detected. The main reason is that unique execution patterns are rare (averagely 2% according to Section 4.1.2), making the hash table sparse and hard to have collisions.

4.4 DISCUSSION

Compatibility to Other Advanced Fuzzers. SAND does not touch the main fuzzing logic of a CGF. It is orthogonal to many other fuzzer advances. For example, new mutation strategies [4, 95], effective seed scheduling schemes [9, 7], and hybrid fuzzing techniques [58, 137] can all be normally integrated into a CGF fuzzer, which SAND can further build upon. At a high level, in the sequence of all mutated inputs during fuzzing, SAND's effectiveness depends on the fact that bug-triggering inputs are mostly likely to have unique execution patterns. Therefore, different mutation strategies may affect SAND's performance. To understand SAND's general applicability, we port it to an alternative fuzzer MOpt [95], which uses a different mutation scheduling strategy and can be manually turned on in AFL++. We include the details in the appendix.

Alternative Test Oracles. Sanitizers essentially provide test oracles for executions. These test oracles are customized for security vulnerabilities. In practice, many other test oracles exist for different application scenarios. For instance, when fuzzing Javascript JIT compilers [5], a semantic correctness test oracle is usually provided. Differential test oracles are applied to find incorrect outputs, such as wrong implementations [119] and platform-dependent divergences [164]. All these test oracles are usually expensive. Applying our idea to selectively feed inputs into the costly test oracle checkers could be beneficial and requires further verification and exploration.

Incompatibility to Coverage-guided Tracing. Our current execution pattern is collected from the coverage bitmap. Some research efforts are trying to reduce coverage collection overhead, such as HexCite [106] and UnTracer [105]. Such approaches break the coverage map and thus cannot be used together with SAND. However, we would like to highlight that the coverage collection overhead is much smaller compared to sanitizers. Researchers [149] have shown that the latest coverage collection approach used in AFL++ only brings a median of 15% overhead. Sanitizers like ASan and MSan can incur 237∼6,836% overhead. Even if these approaches can entirely eliminate coverage tracing overhead, the overall benefit when sanitizers are used is small.

Limitations. Despite that SAND brings significant improvement to fuzzing, it also comes with a few limitations. The first limitation is the gap between the unique execution pattern ratio and the bug-triggering input ratio. Our empirical evaluation in Section 4.1.2 has shown that many bug-triggering input ratios are below 0.5%, which is lower than the average unique execution pattern ratio of 2%. This gap indicates that there is still space for improvement. Designing more effective execution abstraction is an interesting future work. The second limitation is that although our evaluation has confirmed that SAND did not miss any bugs, we can not provide a theoretical guarantee. It would be interesting and useful to explore sound execution analysis to eliminate this concern.

4.5 RELATED WORK

Reducing Sanitizer Overhead. Some research efforts exist to reduce sanitizer overhead. ASAP [143] removes sanitizer checks to meet a required performance budget. FuZZan [62] dynamically selects the most optimal metadata structure for both ASan and MSan to reduce sanitization overhead fuzzing. SanRazor [163] and Debloat [165] remove redundant sanitization checks via either static or dynamic analysis. SanRazor supports both ASan and UBSan while Debloat only supports ASan. All of these techniques require significant modifications to sanitizer implementations, which inevitably hinders their practical adoption. SAND, on the other hand, uses sanitizers without any modification. This feature further brings the orthogonality of SAND to these efforts. For instance, we can replace the ASan-enabled program with Debloat-enabled program to benefit from the improvement of Debloat.

Bug Pattern. Igor [63] observes that all bug-triggering inputs have some unusual execution behavior. For specific bug types, UAFL [145] intuitively prioritizes memory operations of longer sequences to detect User-After-Free bugs effectively. Dowser [51] selectively checks instructions that access arrays in a loop for discovering buffer overflow bugs. ParmeSan [116] leverages the knowledge from sanitizer instrumentations to discover certain types of bugs faster. Our PGE [76] finds that bug-triggering executions correlate with execution prefixes. At a high level, the findings or insights behind these approaches share similar motivations to our execution pattern, *i.e.*, bug-triggering inputs tend to have unique execution features.

Improving Fuzzing Performance. These related efforts have been previously discussed in Section 3.5 of Chapter 3.

Part II

RELIABILITY OF CODE ANALYSIS

5 FINDING BUGS IN SANITIZER IMPLEMENTATIONS

Sanitizers are crucial in enabling the large-scale detection of security vulnerabilities caused by certain undefined behaviors [131, 132], such as buffer overflow, use after free, *etc.* The most commonly used sanitizers include Address Sanitizer (ASan) [130] for memory access errors, Undefined Behavior Sanitizer (UBSan) [88] for various undefined behaviors, and Memory Sanitizer (MSan) [136] for uninitialized memory uses. While substantial research and engineering efforts have been made toward devising efficient fuzzers [105, 106] and reducing sanitizer costs [62, 163, 165], the reliability of sanitizers — essential for detection effectiveness — have received little attention from both academia and industry.

In this chapter, we aim to validate sanitizer implementations by detecting bugs in them. Specifically, we are interested in false negative (FN) bugs, where sanitizers fail to report undefined behaviors in the target program due to their own implementation issues. False negative bugs in sanitizers can significantly break their overall detection effectiveness and thus be considered highly harmful. Despite its criticality and importance, to the best of our knowledge, no work has systematically investigated this problem.

The work in the chapter was published in [78].

Key idea. We introduce the first effective testing framework, UBfuzz, for finding FN bugs in sanitizers. At a high level, the general testing workflow is (1) generating a UB program, *i.e.*, a program exhibiting undefined behavior, and (2) compiling it with sanitizers and executing the compiled binary. If no sanitizer report on a UB program is produced, a potential sanitizer bug is detected. Two main challenges exist, which we will discuss next.

Challenge 1: UB program generation.
In order to detect FN bugs, abundant and diverse UB programs should be available. The automated generation of valid programs for compiler testing has been extensively researched. Tools like Csmith [159] can generate a wide variety of valid C programs that are free from UB. However, the generation of programs exhibiting various types of UB, which is essential for sanitizer testing, remains unexplored. For instance, to test ASan, programs with memory safety bugs such as buffer-overflow, use-after-free, use-after-scope, *etc.*, are needed. One might consider randomly mutating a valid program,

for example, deleting statements or altering variable values, to introduce UBs. As we will demonstrate in our evaluation in Section 5.3.3, this naive mutation-based method is ineffective in generating UB programs—most of the mutated programs do not have UB. Furthermore, the generated programs encompass only a few UB types and are unable to find any FN bugs in sanitizers.

Our solution: Shadow Statement Insertion.
We propose a general approach for introducing UB into a valid program. Given a program and a target UB such as buffer overflow, our approach first applies static and dynamic analysis to learn the program's runtime state and identify a specific program location where the target UB can be introduced. Subsequently, we insert a new statement into the program such that the chosen program location triggers a UB. We term our approach *shadow statement insertion*. For instance, the original program of Figure 1.4 does not have line 7 and is thus free of UB. To introduce a buffer overflow, our tool analyzes the code and identifies that the pointer `d` points to the stack buffer `b` of size 8 bytes. It then inserts `k=2` to overflow the buffer access at line 8. As will be detailed in Section 5.2, following the same framework, our design can be generalized to other UB types.

Challenge 2: Compiler optimization significantly complicates sanitizer testing.
Given an input UB program, a natural approach is to examine whether a compiler's sanitizer such as "`gcc -O2 -fsanitize =address`" can detect the UB. If no report is produced, one might assume that "a sanitizer FN bug is discovered". However, this is not true due to compiler optimizations.

Sanitizers are implemented as passes in the compilers' pipeline. Figure 5.1 illustrates the high-level pipeline in GCC compilation with ASan enabled. The ASan pass collaborates with other optimizer passes to compile a program. Previous research [61, 152] has shown that compiler optimizers always presume that the input program does not contain UB, resulting in the elimination of certain UBs by optimizer passes. Figure 5.2 provides an example where both `d[1]=1` at line 4 and `*b` at line 5 trigger stack-buffer-overflow UB. Nonetheless, if we compile it with ASan at -*O*2 (`gcc -O2 -fsanitize=address`), no UB report will be produced. *Different from the previous example in Figure 1.4, this is not a sanitizer bug.* The reason is that early optimization passes at GCC -*O*2 optimize away all the UB code, as depicted on the right side of Figure 5.2. Since there is no UB present in the input IR to the ASan pass, ASan cannot uncover the UB in the source code. As there is no UB in the final compiled binary, ASan is not considered buggy in this example.

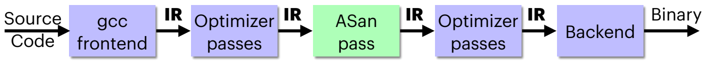


Figure 5.1: The high level compilation pipeline of ASan in GCC/gcc.

```
static int d[1]={0};
int main() {
    int *b = &d[1];
    d[1] = 1;
    return *b;
}
```

GCC -O2 ⟹

```
~~static int d[1]={0};~~
int main() {
    ~~int *b = &d[1];~~
    ~~d[1] = 1;~~
    return ~~*b~~ 1;
}
```

Figure 5.2: GCC -*O*2 optimizes away the UB code, thus ASan cannot discover the UB.

A natural follow-up question is *can we only consider unoptimized compilers such as with -O0?* The answer is no for two main reasons. First, even with -*O*0, some basic optimizations, such as constant folding, may still optimize away the UB code. Second, many sanitizer FN bugs only exist at higher optimization levels, as demonstrated in Figure 1.4. Testing sanitizers only at -*O*0 may fail to detect many critical FN bugs. In practice, a lot of existing real-world use cases use sanitizers with optimizations to obtain high binary execution speed. For example, Google Chrome recommends turning on sanitizers with optimizations in release mode. The OSS-Fuzz project [49], which is a continuous fuzzing platform for open-source projects and has more than 8k stars on GitHub as of today, by default uses sanitizers with -*O*1 to fuzz all projects.

Similarly, differential testing across different compilers is ineffective as it is impossible to determine whether a discrepant report is caused by a sanitizer FN bug or merely due to compiler optimizations. For instance, although GCC ASan at -*O*0 and -*O*2 produce different results in both Figure 1.4 and 5.2, the latter is caused by compiler optimizations.

Our solution: Crash-site mapping as the test oracle.
We introduce a novel test oracle, *crash-site mapping*, to accurately discern whether discrepant sanitizer reports stem from sanitizer FN bugs or compiler optimizations. Given two binaries, b_c and b_n, compiled by two compilers such as GCC ASan at -*O*0 and -*O*2, executing b_c results in a crash while executing b_n exits normally. Here, the crash of b_c means that the sanitizer

successfully reports the UB, while the normal exit of b_n means that the sanitizer does not report the UB. Our primary approach involves using a debugger to trace the execution of both binaries. If the crash location in b_c is also executed by b_n, we can infer that the compiler does not eliminate the UB, and thus, it is highly probable that a sanitizer FN bug is present. A more detailed example will be provided in Section 5.1. As our evaluation will demonstrate, our crash-site mapping oracle can effectively identify discrepancies caused by compiler optimizations.

We realized our solutions in a tool named UBFuzz. It can automatically generate a substantial number of UB programs and accurately identify sanitizer FN bugs. The example we showcased in Figure 1.4 was found by UBFuzz. We reported this FN bug to the GCC team, who confirmed and fixed it. The root cause is that in some cases, GCC ASan would "forget" to insert checks to specific memory accesses, thus resulting in missed UB reports. During a five-month testing period, UBFuzz uncovered a total of 31 new FN bugs in ASan, UBSan, and MSan from both GCC and LLVM.

Main contributions. In summary, we make the following contributions:

- We introduce a general approach, *Shadow Statement Insertion*, for generating UB programs.
- We design *crash-site mapping* as an effective oracle for sanitizer testing.
- Based on the proposed UB generator and test oracle, we develop the automated tool UBFuzz for testing sanitizers.
- We report our extensive evaluation of UBFuzz, which successfully identified 31 sanitizer FN bugs.

The artifact for UBFuzz, including all source code and data, is permanently available [75].

5.1 ILLUSTRATIVE EXAMPLES

This section illustrates (1) how UBFuzz generates UB programs for a target UB, and (2) how our crash-site mapping test oracle works.

5.1.1 *UB Program Generation*

The uncolored (black) code in Figure 5.3 is the seed program. We now demonstrate how we mutate this seed program to the UB program in

```
1 struct a { int x };
2 struct a b[2];
3 struct a *c = b, *d = b;
4 int k = 0;
5 int main() {

    LOG_BufRange(&b[0], sizeof(b));①

6   *c = *b;

    k = 2; ③
    LOG_BufAccess(d+k);②

7   *c = *(d+k);
8   return c->x;
9 }
```

Figure 5.3: Code instrumentation for UB insertion.

Figure 1.4, with the goal of introducing a *stack-buffer-overflow* UB. Our generator works as follows:

Step 1. Insert profiling statements for all stack buffers. Since there is only one global stack buffer `b[2]` in this seed program, a single profiling statement is inserted as indicated by ①. Next, identify all code constructs with the potential to exhibit the target UB. Given that our target UB is stack-buffer-overflow, we statically locate all memory accesses. All the pointer dereferences `*b`, `*c`, and `*(d+k)` at lines 6 and 7 are eligible. For the simplicity of presentation, we only show the profiling statement ② for `*(d+k)`.

Step 2. After the instrumentation, we compile and execute the code to obtain its runtime information, including the memory range of buffer `b` and the memory address of `d+k`.

Step 3. To introduce a stack-buffer-overflow for `*(d+k)` at line 7, we mutate the value of `k` such that it overflows the pointed-to buffer. Since we have learned that the buffer `b` has a size of 8 bytes and `d` points to the starting location of `b`, we insert the shadow statement `k=2` ③ to introduce a stack-buffer-overflow at line 7. One might question whether setting `k` to an arbitrarily large enough value would also introduce a buffer overflow. However, due to the design limitation of ASan, it can only detect overflows of up to 32 bytes. Consequently, the valid range of overflowed addresses

```
...              #line,offset
andl  %r8d, %esi #10,8
movq  %rax, %rdi #10,8
callq 0x10e0     #10,8
```

a. The last three executed instructions in b_c.

```
movq  ptr(%rip),%rax #10,3
movl  $0xfff,(%rax)  #10,8
```

b. The executed instructions are from line 10 in b_n.

Figure 5.4: Partial executed instructions in b_c and b_n. The comment shows the corresponding line and offset in the source code.

falls between $8 \sim 32$ bytes beyond `b`. Our precise runtime analysis enables us to precisely mutate `d+k` such that it falls within this range.

Finally, all logging statements ①② are removed while the shadow statement ③ is kept. The resulting UB program is identical to the one presented in Figure 1.4.

5.1.2 *Crash-Site Mapping as the Test Oracle*

After a UB program is generated, we use at least two compilers with sanitizer enabled to compile it. We then execute the compiled binaries to examine whether there is a discrepant report. If one of the binaries crashes while the other does not, we need to determine if the discrepancy is caused by compiler optimizations. We refer to the crashing binary as b_c and the non-crashing binary as b_n. For the code snippet in Figure 5.3, b_c is compiled by GCC ASan at -O0, while b_n is from -O2. Our crash-site mapping works as follows:

Step 1. *Analyze b_c:* We utilize a debugger[6] to track the execution of b_c and obtain the last executed site, *i.e.*, the crash-site. Figure 5.4a shows the last three executed instructions of b_c, the last of which indicates the crash-site is at (line 10, offset 8). This means that executing the instruction compiled from (line 10, offset 8) in the source code results in a crash, *i.e.*, a sanitizer report.

Step 2. *Analyze b_n:* Once again, we use the debugger to track the execution of b_n. Since we have learned the crash-site in Step 1, we only need to monitor if the crash-site is also executed in b_n. Figure 5.4b shows a part of

6 In our implementation, we use LLDB and its Python API to automate our analysis. More details in the evaluation section.

Table 5.1: UB conditions and shadow statements. The first three columns describe the conditions for certain code constructs not to have the target UB. The fourth column demonstrates the location where our shadow statements will be inserted. The fifth column presents the effect of each shadow statement. The last column lists the instantiation of each shadow statement in our implementation. Here, $x, \widehat{x}, y, \widehat{y}, \widehat{c}$ are $(n+1)$-bit integers; p, q are pointers; a is an array with capacity $\textsc{ArraySize}(a)$; $lhs \xrightarrow{val} rhs$ represents the value of lhs is rhs.

UB	Code Constrcut	Sufficient condition for not having the UB	Shadow Statement $\Delta(\cdot)$	Effect of $\Delta(\cdot)$	Instantiation
Buf. Overflow (Array)	$a[x]$	$0 \le x < \textsc{ArraySize}(a)$	$\Delta(x)$; `Stmt{`$a[x]$`};`	$x \xrightarrow{val} v$ and $(v < 0 \vee v \ge \textsc{ArraySize}(a))$	$\widehat{x} = v - x;$ `Stmt{`$a[x+\widehat{x}]$`};`
Buf. Overflow (Pointer)	$*p$	$p \in \textsc{BufferRange}(p)$	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} q$ and $q \notin \textsc{BufferRange}(p)$	$\widehat{c} = q - p;$ `Stmt{`$*(p+\widehat{c})$`};`
Use After Free	$*p$	$\forall free(q)$, !alias(p,q)	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} q$ and q is freed	$free(p);$ `Stmt{`$*p$`};`
Use After Scope	$*p$	$\textsc{Scope}(*p) \in \textsc{Scope}(p)$	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} q$ and $\textsc{Scope}(*q)$ out of $\textsc{Scope}(p)$	$p = q;$ `Stmt{`$*p$`};`
Null Ptr. Deref.	$*p$	$p \ne \textsc{Null}$	$\Delta(p)$; `Stmt{`$*p$`};`	$p \xrightarrow{val} \textsc{Null}$	$p = 0;$ `Stmt{`$*p$`};`
Integer Overflow	x op y	x op $y \in [-2^n, 2^n - 1]$	$\Delta(x,y)$; `Stmt{`x op y`};`	$x \xrightarrow{val} v_0, y \xrightarrow{val} v_1$ and v_0 op $v_1 \notin [-2^n, 2^n - 1]$	$\widehat{x} = v_0 - x, \widehat{y} = v_1 - y;$ `Stmt{`$(x+\widehat{x})$ op$(y+\widehat{y})$`}`
Shift Overflow	$x \ll y$ or $x \gg y$	$0 \le y < n$	$\Delta(y)$; `Stmt{`$x \ll y$`};`	$y \xrightarrow{val} v$ and $v < 0 \vee v \ge n$	$\widehat{y} = v - y;$ `Stmt{`$x \ll (y+\widehat{y})$`};`
Divide by Zero	x/y or $x\%y$	$y \ne 0$	$\Delta(y)$; `Stmt{`x/y`};`	$y \xrightarrow{val} 0$	$\widehat{y} = -y;$ `Stmt{`$x/(y+\widehat{y})$`};`
Use of Uninit. Memory	`if(`x`)` or `while(`x`)`	x is uninitialized	$\Delta(x)$; `Stmt{`x`};`	$x \xrightarrow{val}$ uninit. memory	int $\widehat{x}$; `Stmt{`$x+\widehat{x}$`};`

```
int a[5]; int x=1;        int a[5]; int x=1;
a[x] = 1;          ⟹      x = 5; //Δ(x);
                          a[x] = 1;
```

Figure 5.5: The expression $x = 5$ is inserted as the shadow statement to introduce a buffer overflow in $a[x]$.

the execution in b_n. We can observe that (line 10, offset 8) is executed as well.

Step 3. *Mapping:* Since the crash-site from b_c is also executed in b_n, we classify this program as triggering a sanitizer FN bug. Otherwise, the discrepancy would be classified as being caused by compiler optimizations.

For the program shown in Figure 5.2, the crash site from GCC ASan-*O*0 is not present in GCC ASan-*O*2, which indicates that the inconsistent sanitizer reports are caused by compiler optimizations. Our evaluation in Section 5.3.4 will demonstrate that crash-site mapping can accurately identify discrepancies resulting from compiler optimizations.

5.2 APPROACH

We first analyze sufficient conditions for a valid program to be free from UB, which motivates the design of our shadow statement insertion method. Then, we introduce the proposed UB program generation approach. Finally, we present the crash-site mapping as the test oracle for sanitizer testing.

5.2.1 *UB Conditions and Shadow Statement*

Code constructs that are free of UB. To generate UB programs, we need first to understand how UB is triggered. The first three columns in Table 5.1 list the conditions for certain code constructs to *not have* the UB, as specified in the C standard [14]. For instance, the first shown UB is buffer-overflow. For an array access $a[x]$ to be free from this UB, the index x should be positive and less than the array size. Another example is signed integer overflow. As long as the calculation of x `op` y falls within the range of $[-2^n, 2^n - 1]$, it is free from this UB. We can conclude that for a code construct that has the adventure of a UB, as long as the given condition is met, it is free from the UB.

Code constructs that have UB. Since we now understand the conditions for having UBs in certain code constructs, we can simply find a way to break the condition to introduce a UB. In this thesis, we utilize *shadow statements* to achieve this purpose. For a valid program $\mathcal{P}$ that contains a code construct *expr*, we introduce a UB by placing a shadow statement `Δ(expr)` before the code construct as follows:

```
Δ(expr);
Stmt{expr};
```

The shadow statement `Δ(expr)` is designed to change the evaluation value of *expr* such that when executing `Stmt{expr}`, the UB condition is triggered. The fourth and fifth columns in Table 5.1 list the shadow statements and their effects. For example, to introduce a buffer overflow to $a[x]$, the inserted shadow statement `Δ(x)` changes the value of x to v, which is out of the range of array a. Figure 5.5 illustrates a concrete example where the shadow statement $x = 5$ is inserted before the array access.
There are two key questions. The first is how to understand the target effect of `Stmt{expr}`. In the above example, we need to know the concrete range of array a, which our generator uses dynamic analysis to obtain. The second is how to instantiate the shadow statement. Once we know the target effect of `Stmt{expr}`, there are plenty of ways to instantiate it. In the above example, we can also choose $x = x + 4$ or $x = x * 4 + 1$ as the shadow statement, which results in the same effect as $x = 5$. The last column in Table 5.1 lists the instantiations we used in our implementation. Details will be discussed next.

5.2.2 *UB Program Generator*

Algorithm 7 shows the general process of generating UB programs. Given a seed program $\mathcal{P}$ and an associated input $\mathcal{I}$, our goal is to generate UB programs that contain the target UB type $\mathcal{U}$ on the input $\mathcal{I}$. Our generator works as follows:

Step 1. *Expression Matching* (line 2): find all expressions in $\mathcal{P}$ that have the target code constructs for the given UB. For example, given buffer overflow, according to Table 5.1, this procedure will find all array accesses *i.e.*, $a[x]$, and pointer dereferences *i.e.*, $*p$.

Step 2. *Program Profiling* (line 3): instrument and run $\mathcal{P}$ on the input $\mathcal{I}$ to collect an execution profile that contains the required runtime information such as the allocated buffers and pointer addresses.

Algorithm 7: UB program generation

```
1 procedure Generator(Program $\mathcal{P}$, Input $\mathcal{I}$, UBType $\mathcal{U}$):
      // find all matched expr to a given UB
2     $E \leftarrow$ GetMatchedExpr($\mathcal{P}$, $\mathcal{U}$)
3     $\widehat{prof} \leftarrow$ Profile($\mathcal{P}$, $\mathcal{I}$, $\mathcal{U}$, $E$)          // profiling
4     $P_{UB} \leftarrow [\,]$
5     foreach $expr \in E$ do
          // synthesize a shadow statement
6         $\Delta(expr) \leftarrow$ SynShadowStmt($expr$, $\widehat{prof}$, $\mathcal{U}$)
          // insert the shadow statement
7         $\mathcal{P}' \leftarrow$ Insert($\mathcal{P}$, $\Delta(expr)$ )
          // append the new UB program
8         $P_{UB}$.append($\mathcal{P}'$)
9     return $P_{UB}$
```

Step 3. *Shadow statement synthesis and insertion* (lines 6-7): for each target *expr*, query the execution profile to synthesize a shadow statement and insert it into the seed program to obtain a UB program.

As indicated by the algorithm, our generator has the following features:

- *Target UB type needs to be specified when being invoked.* For every invocation, our generator will generate a set of UB programs that all have the same target UB type.

- *Only one UB in every generated program.* Lines 6-8 in Algorithm 7 show that for every matched expression, the generator generates a UB program with shadow statement insertion. Consequently, there is a single UB for each generated program.

- *Multiple UB programs for one invocation.* The for loop (line 5) signifies that a UB program is generated for each of the matched expressions. Ultimately, the generator returns a set of UB programs, all containing the same UB type.

Next, we detail each of the above steps.

5.2.2.1 *Expression Matching — GetMatchedExpr(·)*

Given a seed program and a target UB, we statically scan the program to find all expressions that match the code constructs as specified in Table 5.1. For example, suppose our target UB is signed integer overflow. We will find all expressions that have the form of x `op` y, where `op` is an arithmetic operator such as $+$, $-$, and $*$. After scanning, all matched expressions will be saved into E, each item in which contains the matched expression and its location in $\mathcal{P}$.

5.2.2.2 *Program Profiling — Profile(·)*

An execution profile provides runtime information about the program, which is essential for our shadow statement synthesis. We define the execution profile $\widehat{prof}$ as follows.

Definition 6 (Execution Profile)**.** *Given a program $\mathcal{P}$, an input $\mathcal{I}$, and the target expression list E, the execution profile $\widehat{prof}$ records the following information during running $\mathcal{P}$ with $\mathcal{I}$: (1) all the values of expressions in E observed, and (2) all the allocated and freed stack and heap memory address ranges.*

To facilitate easy access to the execution profile, let e denote $expr$, we define the following queries to obtain concrete information from $\widehat{prof}$:

- $Q_{liv}(\widehat{prof}, e)$: return `true` if e is in the live region; otherwise, return `false`. This information is inferred by checking if e has a value in $\widehat{prof}$. If it does, then it is located in the live region; otherwise, $\widehat{prof}$ is unable to obtain its value.
- $Q_{val}(\widehat{prof}, e)$: return the value of e.
- $Q_{mem}(\widehat{prof}, e)$: e is a pointer or an array. Return the memory range that e points to. If the memory has already been freed, return `false`.
- $Q_{scp}(\widehat{prof}, e)$: return the scope of e. We extend $\widehat{prof}$ with scope information obtained from Clang's LibTooling.

In our implementation, given a new seed program, we first obtain its execution profile $\widehat{prof}$ and then synthesize UB programs. Thus, the profiling overhead for all UB types is identical. This is an implementation choice because when testing sanitizers, UBFUZZ, by default, generates all the supported UB programs for one seed program.

5.2.2.3 *Shadow Statement Synthesis and Insertion — SynShadowStmt(·) & Insert(·)*

For a target UB, the synthesized shadow statement should have the effect as shown in the fifth column in Table 5.1. In theory, there are numerous ways to instantiate a shadow statement. For instance, as previously illustrated in Figure 5.5, expressions like $x = 5$, $x = x + 4$, $x = x * 4 + 1$, and many others, all satisfy the requirement. In our implementation, for each shadow statement, we choose the simplest instantiation to minimize changes to the seed program. The last column in Table 5.1 lists the instantiations. Details are as follows:

- *Buffer overflow (array)*: We introduce an auxilary variable $\widehat{x}$ to the original expression to obtain $a[x + \hat{x}]$. $\Delta(expr)$ is $\widehat{x} = v - x$. The value of v is obtained by calculating the memory size from $Q_{mem}(\widehat{prof}, a)$; the value of x is obtained via $Q_{val}(\widehat{prof}, x)$. This instantiation does not change any other program semantics except for our target expression.

- *Buffer overflow (pointer)*: Similarly, we first introduce an auxilary variable $\widehat{x}$ to obtain $*(p + \widehat{x})$. $\Delta(expr)$ is $\widehat{x} = q - p$, where values of q and p are obtained via $Q_{mem}(\widehat{prof}, p)$ and $Q_{val}(\widehat{prof}, p)$, respectively.

- *Use after free*: $\Delta(expr)$ is $free(p)$.

- *Use after scope*: $\Delta(expr)$ is $p = q$, where $Q_{scp}(\widehat{prof}, *q)$ is not within the scope of $Q_{scp}(\widehat{prof}, p)$.

- *Null pointer dereference*: $\Delta(expr)$ is $p = (void*)0$.

- *Integer overflow*: We first introduce auxiliary variables $\widehat{x}$ and $\widehat{y}$ to the original expression to obtain $(x + \widehat{x})$ `op` $(y + \widehat{y})$. Then the shadow statement is set to $\widehat{x} = v_0 - x, \widehat{y} = v_1 - y$. The values of x and y are obtained via $Q_{val}(\widehat{prof}, x)$ and $Q_{val}(\widehat{prof}, y)$. To find the proper values v_0 and v_1, we adopt Monte Carlo to sample from $[-2^n, 2^n - 1]$ such that $(x + \widehat{x})$ `op` $(y + \widehat{y})$ exceeds the range of an $(n + 1)$-bit integer.

- *Shift overflow*: We first introduce an auxiliary variable $\widehat{y}$ to obtain $x \ll (y + \widehat{y})$ or $x \gg (y + \widehat{y})$, and then set the shadow statement to $\widehat{y} = v - y$. The value of y is obtained via $Q_{val}(\widehat{prof}, y)$ and v is a random value satisfying $v < 0 \lor \geq n$.

- *Divide by zero*: We first introduce an auxiliary variable $\widehat{y}$ to obtain $x/(y+\widehat{y})$, and then set the shadow statement to $\widehat{y} = -y$. The value of y is obtained via $Q_{val}(\widehat{prof}, y)$.
- *Use of uninitialized memory*: We first introduce an auxiliary variable $\widehat{x}$ to obtain $x + \widehat{x}$, and then set $\Delta(expr)$ to `int` $\widehat{x}$. Since $\widehat{x}$ is uninitialized, $x + \widehat{x}$ becomes uninitialized as well.

Some of the above operations, such as *Buffer overflow (pointer)*, need to know the precise pointer information to synthesize shadow statements accurately. Such pointer information can be obtained via $Q_{mem}(\widehat{prof}, e)$, which achieves this goal by logging all allocated pointers' addresses and used pointers (see the example in Section 5.1.1 **Step 1**). Thus, we do not need any separate pointer analysis.

5.2.2.4 *Discussions*

As the evaluation will demonstrate, our generator can effectively generate interesting UB programs for sanitizer testing. Despite its effectiveness, it also comes with certain limitations. First, our UB program generator relies on seed programs. If seed programs are not expressive enough, UBFUZZ cannot generate useful UBs. Fortunately, seed program generators like Csmith have proven effective in exercising rich language features [66, 159]. Second, the list of UB in UBFUZZ is non-exhaustive. The C17 standard [14] lists 219 UB types. Not all UBs are supported by sanitizers. We selected UBs that are (1) supported by at least one sanitizer and (2) included in the CWE list [113], which enumerates all common weaknesses in C by the MITRE community. Our supported UBs cover all UBs studied by the related work [61, 152]. Generally, each UB comes with a root cause and can be represented in a generic pattern, as demonstrated in our approach. For instance, using pointer subtraction to determine size is UB if two pointers point to different objects [110]. Realizing this UB in UBFUZZ would require knowledge of the address ranges of each object and pointer, which can be easily obtained through dynamic profiling. We chose not to realize this UB because none of the existing sanitizers support its detection.

For each seed program, as a new UB is introduced, its semantics is consequently altered. We clarify that preserving the seed program's semantics is not necessary in our application scenario because we only require the resulting program to contain the desired UB. Second, all UB programs have invalid, often nondeterministic semantics because (1) their semantics rely on how the compiler deals with UBs, and (2) the compiler has full

Algorithm 8: Crash-Site Mapping

```
procedure IsBug($Binary\ b_c$, $Binary\ b_n$):
    $S_c \leftarrow$ GetExecutedSites($b_c$)
    $S_n \leftarrow$ GetExecutedSites($b_n$)
    if $S_c[-1] \in S_n$ then
        return True
    else
        return False
procedure GetExecutedSites($Binary\ b$):
    $S \leftarrow [\,]$
    debugger.Init($b$)
    while debugger.IsAlive() do
        $l \leftarrow debugger.curr\_line$
        $o \leftarrow debugger.curr\_offset$
        $S$.append(($l, o$))
        debugger.NextInstruction()
    return $S$
```

freedom in handling code with UBs. Nevertheless, UBFuzz still preserves the runtime semantics of a seed program up to the mutation site.

5.2.3 *Crash-site Mapping as the Test Oracle*

With the generated UB programs, we employ differential testing across multiple compilers to find sanitizer FN bugs. Without loss of generality, assume that we have two compilers $\mathcal{C}_c$ and $\mathcal{C}_n$ with the same sanitizer enabled, *e.g.*, GCC ASan at -O1 and LLVM ASan at -O1. The corresponding compiled binaries are b_c and b_n. Suppose that executing b_c results in a crash while b_n exits normally. Here, the crash in b_c means that the sanitizer in $\mathcal{C}_c$ *successfully* reports the UB; the normal exits of b_n means that the sanitizer in $\mathcal{C}_n$ *does not* report any UB. As analyzed at the beginning of this chapter, the discrepancy can arise from a sanitizer FN bug or merely compiler optimizations. Our *crash-site mapping* can identify the true cause of the discrepancy. Before introducing our approach, we formally define *crash site*.

Table 5.2: UB types supported by each sanitizer.

UB	Sanitizer
Buf. Overflow(Array)	ASan, UBSan
Buf. Overflow(Pointer)	ASan
Use After Free	ASan
Use After Scope	ASan
Null Ptr. Deref.	UBSan
Integer Overflow	UBSan
Shift Overflow	UBSan
Divide by Zero	UBSan
Use of Uninit. Memory	MSan

Definition 7 (Crash Site). *A binary b_i is compiled from program $\mathcal{P}$ and running b_i results in a crash. We denote the last executed instruction as $\widehat{inst}$. If $\widehat{inst}$ corresponds to the line l and offset o in $\mathcal{P}$, then the crash site of b_i is (l, o).*

Our key insight is that if the crash site in b_c is also executed by b_n, the compiler $\mathcal{C}_n$ does not optimize away the UB-triggering expression in $\mathcal{P}$; thus the discrepancy is caused by a sanitizer FN bug in $\mathcal{C}_n$. Algorithm 8 details our approach.

We first obtain the executed sites of both b_c and b_n, *i.e.*, all the executed (line, offset) in $\mathcal{P}$ (line 2-3). If the last executed site in b_c, *i.e.*, the crash site, is also present in b_n's executed sites, return true (line 4-5). Otherwise, return false (line 7). To obtain all executed sites in a binary, we utilize a debugger to track the execution. The procedure GetExecutedSites() provides the necessary steps. Note that when the debugger reaches an instruction, the *debugger.curr_line* and *debugger.curr_offset* return the line and offset in the source program that the instruction corresponds to. The effectiveness of crash-site mapping depends on its accuracy in identifying discrepancies caused by compiler optimizations. Our evaluation in Section 5.3.4 will show that it can achieve near-perfect accuracy.

5.3 EMPIRICAL EVALUATION

Our evaluation is based on the following research questions:

RQ1 **Bug-finding:** Is UBFUZZ effective in finding FN bugs in sanitizers?

RQ2 **UB generator:** How effective is our UB program generator in constructing interesting UB programs?

RQ3 Crash-site mapping: How accurate is the crash-site mapping test oracle in identifying discrepancies caused by compiler optimizations?

RQ4 Code coverage: Can UBFuzz improve code coverage?

5.3.1 *Implementation and Evaluation Setup*

Implementation. Our realization of UBFuzz consists of ∼2,000 lines of C++ and ∼4,400 lines of Python. We use Clang's LibTooling [81] to implement expression matching in Section 5.2.2.1 and program instrumentation for execution profiling in Section 5.2.2.2. We utilize LLDB [87] as the debugger in crash-site mapping and use its Python API to automate the analysis process. Our UBFuzz can run in a fully automated manner in testing sanitizers, including UB program generation, crash-site mapping, and debugging procedures. Once launched, our tool will automatically generate UB programs and use the crash-site mapping algorithm to find FN bugs.

Compilers and sanitizers. Sanitizers are integrated into compilers. We used UBFuzz to test the latest development versions of both GCC and LLVM, which support the most widely-used sanitizers, namely ASan, UBSan, and MSan. Note that MSan is not yet supported by GCC. Since sanitizers are typically used with optimizations, we enabled the most frequently used optimization levels, namely *-O0*, *-O1*, *-Os*, *-O2*, and *-O3*, in both compilers for differential testing.

Seed programs. We use Csmith [159] — a random C program generator — to produce valid seed programs. There are three main reasons:

(1) Csmith is adopted by a lot of compiler testing work [139, 141, 150] and has become the de facto default program generator in testing C compilers;

(2) Csmith can generate complex programs with rich features (*e.g.*, pointer and integer operations), thus offering UBFuzz abundant opportunities to generate diverse UB programs and

(3) programs generated by Csmith are self-contained, meaning that they do not take inputs and can be executed.

Hardware. We conducted all our evaluations on two Linux servers running Ubuntu 20.04 LTS. Both are equipped with an AMD EPYC 7742 64-Core CPU and 256GB RAM.

Table 5.3: Status of the reported bugs in GCC and LLVM.

Status	**GCC**		**LLVM**			**Total**
	ASan	UBSan	ASan	UBSan	MSan	
Reported	9	7	6	8	1	31
Confirmed	8	7	2	2	1	20
Fixed	3	3	0	0	0	6
Invalid	1	0	0	0	0	1

Testing process. Our testing process is fully automated and runs continuously. We first use Csmith to generate a well-formed seed program. Then, for each of the supported UB, we apply UBfuzz to generate UB programs from the seed. For each of the UB programs, as we know their UB type, we use compilers with the corresponding sanitizer enabled to compile and run it. Table 5.2 lists the supported sanitizers for each UB. Once a discrepancy is found, we apply crash-site mapping to decide if it is a sanitizer FN bug. If so, we use C-Reduce to reduce the UB program and report the reduced program to the respective bug tracker. During a period of five months, we sporadically tested the sanitizers. UBfuzz generated around 130 million UB programs. Note that since our work focuses on in-house testing, we assume no adversary is present. Thus, successful sanitization always results in a crash.

5.3.2 *RQ1: Bug Finding*

Table 5.3 summarizes the sanitizer bugs we discovered during our testing period. Overall, we reported 31 bugs. The developers have confirmed 20 of them as previously unknown, real bugs. This highlights the significant bug-finding capability of UBfuzz. Of all these bugs, 6 of them have been fixed and all the fixed bugs are in GCC. The relatively high number of unfixed bugs could be attributed to the fact that many of the reported bugs are introduced since the launch of sanitizers and affect *all stable compiler versions*. Our later analysis will show this fact. We also experienced that the LLVM developers were less responsive than GCC and mostly only

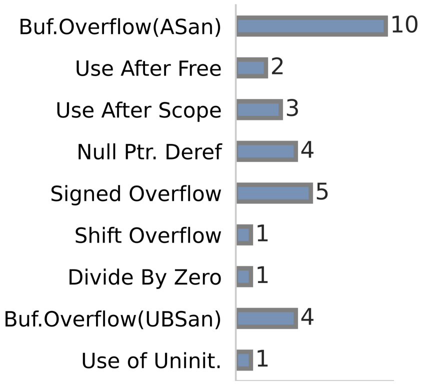


Figure 5.6: Number of bugs triggered by each kind of UB.

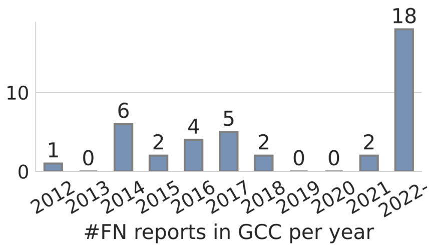
```
1 int a, b;
2 int main() {
3  int *s = &a;
4  for(b=0;b<=3;b++){
5   int i = *s;
6   s = &i;
7  }
8  *s = b;
9 }
```

Figure 5.7: A use-after-scope UB at line 8.

labeled our reports as sanitizer bugs without further diagnosis. We are strict in marking a bug as confirmed — only if the developers have clearly diagnosed it and responded to us. This causes, although UBFUZZ found nearly the same number of bugs in GCC and LLVM, most confirmed and all fixed bugs are found in GCC.

Figure 5.6 shows the number of bugs triggered by each kind of UB. Since both ASan and UBSan support the detection of buffer overflow, we split the found bugs into BufOverflow (ASan) and BufOverflow (UBSan). We can observe that buffer overflow programs triggered the most number of bugs in ASan. Notably, UBFUZZ detected bugs in all UB types, which highlights its strong bug detection capability and the importance of extensively testing sanitizers. Of the 31 bugs, 29 are sanitizer FN bugs, meaning that sanitizers failed to detect UBs in them. Interestingly, we also found two bugs that are not sanitizer FN bugs but rather wrong reports, which means that sanitizers report a UB but with incorrect report information, such as a wrong UB type warning.

➤ **Are there any false alarms by UBfuzz?** We encountered one false alarm report generated by UBFUZZ as indicated by the "Invalid" row in Table 5.3. The reported program is shown in Figure 5.7. It contains a use-after-scope at line 8 because `s` points to an inner scope variable `i`. GCC ASan at *-O3* can not detect it. Our crash-site mapping can verify that line 8 is still present at *-O3*. The GCC developers marked this report as invalid because GCC *-O3* removes the for loop and moves out the inner code, which invalidates



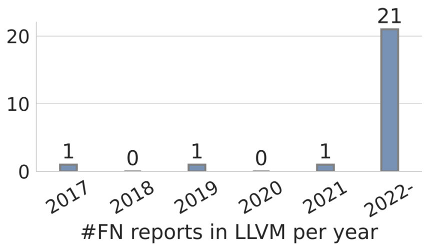


Figure 5.8: Number of sanitizer FN bug reports in GCC and LLVM bug trackers per year.

the use-after-scope UB. This program reveals a limitation of our crash-site mapping test oracle. Nevertheless, the significant number of reported true bugs already demonstrates its effectiveness.

➤ **How significant are the bug-finding results?** To approach this question, we have conducted a manual analysis of all reported false negative bugs based on GCC and LLVM bug trackers of sanitizers. We chose GCC-5 (released in 2015) and LLVM-5 (released in 2017) as the earliest versions because they are the first stable versions that support sanitizers. The results are shown in Figure 5.8. In the past decade, there were a total of 40 false negative reports on GCC's sanitizers. Of these 40 bugs, UBFUZZ found 16 (40%). For LLVM, UBFUZZ found 14 (58%) out of the 24 bugs. As an intermediate conclusion, UBFUZZ has found a significant number of interesting bugs in both GCC's and LLVM's sanitizers. To further understand the influence of our reported bugs in different stable releases of compilers, we also ran the UB programs that accompany our bug reports on all stable compiler versions. Figure 5.9 presents the number of sanitizer bugs that affect each stable compiler version. It indicates that UBFUZZ can find many long-standing latent bugs, further confirming the significance of our bug-finding results.

➤ **Affected optimization levels.** As shown in Figure 5.10, we counted the number of bugs that affect each optimization level. The result reveals that sanitizer bugs affect all optimization levels. Testing only one of the optimization levels, such as *-O0*, would miss many bugs that only appear at other optimization levels. This demonstrates the usefulness of our crash-site mapping test oracle in identifying sanitizer bugs across optimization levels. There is no clear tendency on which optimization levels are more sensitive

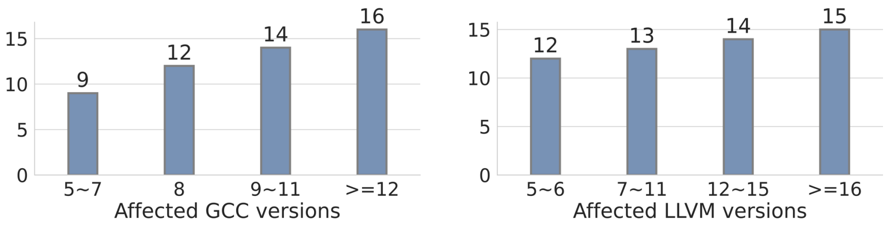


Figure 5.9: Stable compiler versions that are affected by the reported sanitizer FN bugs.

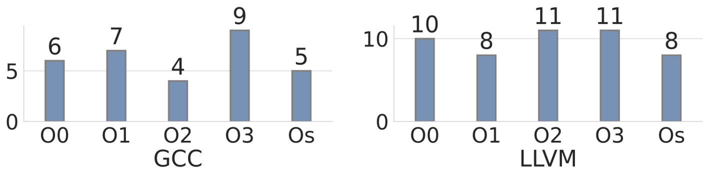


Figure 5.10: Affected optimization levels

to sanitizer bugs. It correlates to the fact that sanitizers and compiler optimizations work independently, as having been shown in Figure 5.1.

5.3.3 *RQ2: Effectiveness of UB Program Generator*

This section provides an in-depth understanding of the effectiveness of our UB program generator. Although there is no other UB program generator that we could compare UBFUZZ against, we use the following two generators as the baseline:

- *MUSIC* [121] is a program mutator designed for mutation testing. It mutates a valid program's abstract syntax tree (AST) to generate syntactically valid mutants. By design, MUSIC may also generate UB programs as it has no guarantees regarding program semantics.

- *Csmith-NoSafe* means that one runs Csmith with its `-no-safe-math` option. To avoid UB at runtime, Csmith utilizes many safe wrappers. For example, it changes all `x/y` to `(y==0?1:x/y)` to avoid division-by-zero. We use its `-no-safe-math` option to disable all the safe wrappers, which may introduce UB in the generated programs.

Table 5.4: The number of generated UB programs per generator. The "No UB" column shows the number of generated programs that do not contain UB. UBFUZZ having "-" on this attribute means that all of its generated programs contain UB.

Generator	**UB**				
	Buf.Overflow (Pointer)	Use After Free	Use After Scope	Null Ptr. Deref	Integer Overflow
UBFUZZ	4,213	3,032	461	2,082	408
MUSIC	27	0	0	1	151
Csmith-NoSafe	0	0	0	0	220

Generator	**UB**					**No UB**
	Shift Overflow	Divide by Zero	Buf.Overflow (Array)	Use of Uninit.	**Total**	
UBFUZZ	287	329	2,396	664	13,872	-
MUSIC	487	3	26	9	704	13,296
Csmith-NoSafe	5,286	1,899	0	0	7,405	6,595

For each generator, we assess the quantity of each type of UB program that the respective generator can produce. We also equip the two baseline generators with the crash-site mapping oracle to test sanitizers.

Generation quantity. We first use Csmith to randomly generate 1,000 seed programs. For each seed program, we use our generator to generate UB programs for every UB type that we support. Table 5.4 details the results. The column "Total" shows that out of the 1,000 seed programs, UBFUZZ generates 13,872 UB programs, averaging 14 UB programs per seed. The generated programs cover all UB types that we support. Buffer overflow takes up the most generated UB programs. The reason is that the seeds from Csmith contain a large number of array and pointer operations, on which UBFUZZ can generate buffer overflow programs. Relatively fewer UB programs are generated on some of the UB types, such as UseAfterScope and DividebyZero. The main reason is that the code constructs they require are stricter than others. For example, DividebyZero can only happen if operators "/" or "%" are present in the live code regions. Comparatively, NullPtrDeref requires only a pointer dereference such as "$*p$", which apparently appears more often.

For a fair comparison to UBFuzz, we apply MUSIC to randomly generate 14,000 programs from the 1,000 seeds used by UBFuzz. Then, we utilize sanitizers[7] to compile and analyze these programs to know if each of them contains UB. Table 5.4 shows that there are only 704 (4%) out of the 14,000 programs containing UB. The other 13,296 (95%) do not contain UB. We now use Csmith-NoSafe to generate programs. Because Csmith-NoSafe does not require a seed program, we directly use it to generate 14,000 programs. Similarly, we use sanitizers to analyze if each of the programs contains UB. From the last row in Table 5.4, we can find that around half (7,405) of the programs contain UB. This number is not as high as UBFuzz but already much better than MUSIC. Notably, all the UB programs are only in three types, *i.e.*, IntegerOverflow, ShiftOverflow, and DividebyZero. This is consistent with how Csmith-NoSafe work: it removes safe wrappers around numeric operations. In summary, UBFuzz can generate the most number of UB programs and cover the most types of UB. Next, we will use MUSIC and Csmith-NoSafe as the UB generator to test sanitizers extensively.

Testing sanitizers with MUSIC and Csmith-NoSafe. To understand if UB programs produced by the baseline generators can also find sanitizer FN bugs, we replace the generator component in UBFuzz with MUSIC and Csmith-NoSafe. The crash-site mapping remains unchanged to serve as the test oracle. During our testing, we let each generator generate around 1 million programs. In the end, *we did not find any sanitizer FN bugs*. The reason for the failure of MUSIC could be that most of the generated programs did not exercise UB. For Csmith-NoSafe, its failure is mainly due to (1) the narrow range of UB types it can generate and (2) unlike our generator, it typically introduces multiple UB in a program, which makes it hard to discover missed sanitizer reports.

Testing sanitizers with the existing UB test suite. The Juliet test suite [109] released by NIST consists of a collection of UB programs. It is by far the most comprehensive test suite for UB detectors. To understand if UB programs from the existing test suite can find sanitizer bugs, we select all the 16,344

7 We run each program with all sanitizers. If a sanitizer reports UB on a program, we use its report to get its UB type. Note that the programs generated by UBFuzz do not need such analysis because the design of UBFuzz allows us to know the UB type of each generated program.

UB programs from the Juliet test suite that are detectable by sanitizers. Instead of using a generator, we directly use all the UB programs from the test suite as the source of programs. Our results show that *none of the UB programs from the Juliet test suite can find sanitizer FN bugs.* This further confirms the necessity of a UB program generator like ours.

5.3.4 *RQ3: Effectiveness of Crash-Site Mapping*

For each generated UB program, we apply differential testing to find discrepancies across compilers. We then use our crash-site mapping to determine if a discrepancy is caused by a sanitizer FN bug or merely compiler optimizations. For the 13,872 UB programs generated from Section 5.3.3, we run all the sanitizers specified in the evaluation setup (Section 5.3.1) to select programs that cause discrepant sanitizer reports. This results in a total of 6,567 selected programs, nearly half of the generated UB programs. The substantial number of discrepancy-causing programs highlights (1) the exceptional quality of our generated UB programs and (2) that without our crash-site mapping, discerning real sanitizer bug-caused discrepancies from the 6,567 discrepancies would be practically infeasible. To evaluate the effectiveness of our crash-site mapping test oracle, we measure its *precision* and *recall*.

Precision: *Out of all selected discrepancies, how many are truly caused by sanitizer bugs?* Out of the **6,567** discrepancies, our crash-site mapping selected **58** and dropped the rest **6,505** as invalid. We manually verify each of the selected discrepancies to see if they are caused by compiler optimizations. *Our manual analysis found that all discrepancies selected by crash-site mapping are due to sanitizer bugs, which means that our crash-site mapping achieves perfect precision.* Although we have analyzed an invalid report by UBfuzz in Section 5.3.2, it does not appear in our quantitative evaluation. Thus, we may conclude that our crash-site mapping has a high precision.

Recall: *Out of all sanitizer bug-caused discrepancies, how many are selected?* This measures if our crash-site mapping will miss interesting discrepancies. Ideally, we should analyze all the dropped discrepancies to verify if any of them are due to sanitizer bugs. However, this requires a manual analysis of 7,966 discrepancies. To reduce the cost, we randomly sampled 200 dropped discrepancies by the crash-site mapping and then manually analyzed each of them. Perhaps surprisingly, after our analysis, we found that none of the dropped discrepancies were caused by sanitizer bugs. In other words, *our*

Table 5.5: Line coverage (LC), function coverage (FC), and branch coverage (BC) of GCC and LLVM.

	GCC			**LLVM**		
	LC	FC	BC	LC	FC	BC
Seeds	63.1%	65.5%	49.4%	30.4%	38.2%	23.3%
MUSIC	63.1%	65.5%	49.4%	30.5%	38.2%	23.4%
Csmith-NoSafe	63.6%	65.5%	50.1%	32.5%	40.2%	24.8%
UBFUZZ	63.7%	65.5%	50.8%	31.8%	39.3%	24.3%

crash-site mapping achieves 100% recall on these samples. Since our evaluation is on sampled data, it is not complete, but it does suggest that crash-site mapping has a high recall.

Soundness of Crash-Site Mapping: As defined in Definition 7, the *crash site* is associated with the source location of the last executed instruction. The soundness of *crash-site mapping* largely depends on a reliable mapping between instructions and source locations. In our implementation, we enable `-g` option for all compilations, which enriches the produced binaries with debugging meta-data. These meta-data can then be utilized by a debugger to obtain the source location, *i.e., (line number, offset)*, of each instruction. Although compiler optimizations can remove instructions with their meta-data, it will not cause the soundness problem in *crash-site mapping* because this resides in the scope discussed in Challenge 2 at the beginning of this chapter. Unfortunately, a recent study [141] has confirmed that bugs in compilers may lead to incorrect debugging meta-data. Buggy meta-data can theoretically cause unsound or incorrect *crash-site mapping* results. For instance, *crash-site mapping* can incorrectly flag the existence of an eliminated crash-site, and thus generate false positive reports. During our extensive testing period, we did not observe any false positive reports, though. We believe such compiler bugs to be rare in our testing scenario. Handling buggy debugging meta-data is an orthogonal research program, and we assume that the correct meta-data is always used in this work.

Table 5.6: Bug category according to root cause analysis.

Category	GCC	LLVM
No Sanitizer Check	2	2
Incorrect Sanitizer Optimization	5	3
Wrong Red-Zone Buffer	1	1
Incorrect Sanitizer Check	2	7
Incorrect Expression Folding/Shorten	4	1
Incorrect Operation Handling	0	1
Wrong Line Information	2	0

5.3.5 *RQ4: Code Coverage*

We utilized Gcov and only instrumented sanitizer-related files to collect coverage in both GCC and LLVM. We used the generated programs from Section 5.3.3 to profile coverage. Table 5.5 summarizes our results. In all cases, compared to the seed programs, all generators lead to a moderate coverage improvement, with UBFUZZ and Csmith-NoSafe showing the largest increase on GCC and LLVM, respectively.

5.3.6 *Case Study*

In order to understand why sanitizers make mistakes, we categorize all bugs according to their root causes. The categorization is based on both our manual analysis and developers' feedback. Table 5.6 shows the result. Both GCC and LLVM make some common mistakes. For example, their sanitizer implementations may conduct "Incorrect Sanitizer Optimization" causing valid sanitizer checks to be removed. We discuss a selection of representative bugs in each bug category.

Figure 5.11a: (*No Sanitizer Check*) This program contains an overflowed memory access at line 7. Since `p_ptr` initially points to pointer `ptr` at line 2, `ptr` will point to the overflowed address `&buf[3]` after line 6. Therefore, a stack-buffer-overflow occurs at line 7, and then the value `0xfff` is written to

```
1 int g, *ptr = &g;
2 int **p_ptr = &ptr;
3 int main() {
4   int buf[3]={1,2,3};
5   *ptr = 1;
6   *p_ptr =&buf[3];
7   *ptr = 0xfff;
8 }
```

a. GCC ASan at -*O*1 missed the buffer overflow access `*ptr` at line 7. [44]

```
1 int a, c;
2 short b;
3 long d;
4 int main() {
5     a = (short)(d == c |
6         b > 9) / 0;
7     return a;
8 }
```

b. GCC's UBSan at all levels missed the DividebyZero at line 5. [46]

```
1 void b() {
2   int c[1];
3   c;
4 }
5 int main() {
6   int d[1]={1};
7   int *e = d;
8   a = 0;
9   for(;a<=5;++a){
10    int f[1]={};
11    e = f;
12    a||(b(), 1);
13  }
14  return *e;
15 }
```

c. GCC's ASan missed the use after scope at line 14, where the pointer `e` points to an inner scope variable `f` defined at line 10. [45]

```
1 volatile int a[5];
2 void b(int x) {
3     if(x)
4     {
5       a[5] = 7;
6     }
7 }
8 int main(){ b(1); }
```

d. LLVM's ASan missed the buffer overflow at line 4. [91]

```
1 int main() {
2     int *a = 0;
3     int b[3]={1, 1, 1};
4     ++b[2];
5     ++(*a);
6 }
```

e. LLVM's UBSan missed the null pointer dereference at line 5. [92]

```
1 int main() {
2     unsigned char a;
3     if (a-1)
4         printf("boom!\n");
5     return 1;
6 }
```

f. LLVM's MSan missed the use of uninitialized memory at line 3. [93]

Figure 5.11: Sample UB programs that trigger sanitizer FN bugs.

`buf[3]`. However, due to a sanitizer instrumentation bug, GCC ASan at -*O*2 fails to insert the check for the validity of `*ptr` at line 7, and thus cannot report it. This bug affects GCC trunk and has been fixed.

Figure 5.11b: (*Incorrect Expression Folding/Shorten*) This program reveals a long latent bug in GCC UBSan, which fails to report the DivisionbyZero UB at line 6. The root cause is that UBSan only cares about integer operands but not booleans. However, although `(d==c|b>9)` is boolean, it gets widened to `short`. GCC UBSan incorrectly handles this case and thus misses the UB. This bug has existed since the introduction of UBSan in GCC.

Figure 5.11c: (*Incorrect Sanitizer Optimization*) This program contains a UseAfterScope UB at line 14, where `e` points to an inner scope variable `f`. GCC ASan fails to report this bug at -*O*3. In fact, GCC ASan initially indeed inserts a scope check for `f` at line 10, but another sanitizer analysis module removes this check when exiting the loop.

Figure 5.11d: (*Wrong Red-Zone Buffer*) This program has an overflowed array access at line 4, where the array `a` is of length `5`. LLVM ASan incorrectly marks the overflow access as within the scope of array padding, while, in fact, it is not. This bug reveals a fundamental problem with ASan handling of global arrays. It affects all LLVM versions at all optimization levels.

Figure 5.11e: (*Incorrect Sanitizer Check*) This program contains a NullPointerDereference UB at line 5, where the pointer `a` is NULL and the program tries to increment it. LLVM UBSan does not report this bug because the null pointer check is not placed before the increment operation. The developer believes that the ++ operator misleads UBSan's internal logic because if we replace `++(*a)` with `*a += 1`, UBSan would work again.

Figure 5.11f: (*Incorrect Operation Handling*) The `if` branch in this program can be taken differently depending on the value of uninitialized variable `a`. LLVM MSan incorrectly handles the subtraction and thinks that the value of `(a-1)` is fully determined. The LLVM developers have confirmed this bug and are working on a fix.

5.3.7 *Discussion on Approach Generality*

Despite the fact that sanitizers are the most popular UB detectors, there are many other dynamic and static UB detection tools. Dynamic tools such as

Dr. Memory [13] and Valgrind [107] can detect memory errors, including buffer overflows, use of uninitialized memory, improper free, *etc.* Static tools such as CppCheck [25] and Infer [104] can detect null pointer dereferences, integer overflows, *etc.* In principle, our approach can also be used to test these detectors. We currently focus on sanitizers because they have a wider real-world impact, especially in the area of fuzzing. Our evaluation results on testing sanitizers have already confirmed the significant UB program generation and bug-finding capability of our tool. Extending our testing scope to other detectors would be an interesting application of our approach and help solidify these additional tools.

5.4 RELATED WORK

Compiler Testing. Finding compiler bugs has been extensively studied; significant research effort has been devoted to testing various compiler functionalities. Many efforts have been put into studying the generation of valid programs [17], which can be used for testing compilers. In the scope of validating sanitizers or bug detectors in general, programs with various defects, however, are required. UBFUZZ can mutate a valid seed program by injecting various undefined behaviors. Therefore, UBFUZZ, in principle, can benefit from the broader program generation research. We defer the discussion of current compiler testing research to Section 6.5 in Chapter 6.

Sanitization. ASan and MSan use shadow memory to record and check the safety of each memory access. Runtime checks are inserted around memory accesses during the compilation of a program. Similarly, UBSan uses tailored checks for different UBs, such as overflow checks for additions and null pointer checks for pointer dereferences. These checks will inevitably increase a program's runtime overhead. Many approaches have been proposed to reduce the overhead by removing redundant checks [163], optimizing checks [165], or applying checks to only a subset of the original code [71, 143]. These optimizations are meaningful in improving sanitizers' practical utility. UBFUZZ can also be used to validate their implementations once they are integrated into mainstream compilers.

Part III

RELIABILITY OF CODE COMPILATION

6 BOOSTING COMPILER TESTING BY INJECTING REAL-WORLD CODE

Even if developers can write fully correct code or apply powerful code analysis tools to eliminate all possible bugs from the code, the resulting binary or executable may still be buggy due to compiler bugs. To improve compilers' reliability, the community has proposed various testing approaches to detect compiler bugs. The most representative techniques are (1) random program generation by featuring a set of generation rules [85, 86, 159] and (2) EMI-based mutation [66, 139] by mutating a random program while preserving its run-time behaviors on specific inputs. However, these rule-based approaches have a common and fundamental limitation: Their expressiveness is constrained by the underlying rules.

Key idea. In this chapter, we introduce a novel approach for testing optimizing compilers with code from real-world applications. The core idea is to construct well-formed programs by fusing multiple code snippets from various real-world projects. The key insight is backed by the fact that the large volume of real-world code exercises rich syntactical and semantic language features, which current engineering-intensive approaches like random program generators are hard to fully support. To construct well-formed programs from real-world code, our approach works by (1) extracting real-world code at the granularity of function, (2) injecting function calls into seed programs, and (3) leveraging dynamic execution information to maintain the semantics and build complex data dependencies between injected functions and the seed program. With this idea, our approach complements the existing generators by boosting their expressiveness via fusing real-world code in a semantics-preserving way.

One may think that ***why can't we use real-world code directly for compiler testing?*** First, real-world code is unlikely to trigger observable compiler issues directly. Marcozzi *et al.* [101] studied the impact of miscompilation bugs with 10 million lines of C/C++ code from 309 Debian packages. They run standard test suites from these packages to identify possible miscompilation bugs. Among dozens of studied miscompilation bugs, only one test failure was observed during a five-month experiment. Such inefficiency is clearly not suitable for stress-testing compilers. Second, as shown in

Chapter 2, real-world code often contains bugs, *e.g.*, undefined behavior. Input programs for compiler testing need to be well-formed. Otherwise, even if a failure, such as incorrect output, is observed, we cannot differentiate whether or not it is caused by undefined behavior or compiler bugs. Numerous efforts [152, 169] in academia and industry have been made to detect and remove undefined behavior from real-world code, yet they still remain unsolved. Last, reducing a project into a simple enough test case suitable for a bug report is challenging. Non-trivial open-source projects typically have a large code base. Reduction is necessary for compiler developers to diagnose and fix bugs. For instance, EMI [66] reported that although they found inconsistent outputs from real-world code, they were not able to reduce them.

We implement our idea in a tool, CREAL, to test C compilers. In a nine-month testing period, we have reported 132 bugs to GCC and LLVM, two of the most popular and well-tested C compilers. At the time of writing, 121 of them have been confirmed as unknown bugs, and 97 of them have been fixed. Most of these bugs were miscompilations, and many were recognized as long-latent and critical. Our evaluation results evidently demonstrate the significant advantage of using real-world code to stress-test compilers. We believe this idea will benefit the general compiler testing direction and will be directly applicable to other compilers.

Main contributions. In summary, we make the following contributions:

- We propose injecting real-world code into seed programs to create diverse, well-formed programs for testing compilers.
- We develop CREAL to implement our idea by injecting real-world functions into seed programs. To fuel CREAL, we build a function database containing over 51,000 real-world functions and
- We conduct a nine-month extensive evaluation of CREAL to demonstrate its effectiveness — 132 unique bugs are discovered in widely used production C compilers: GCC and LLVM.

We believe our idea of injecting real-world code into synthetic programs offers an exciting and promising technical direction for testing C compiler implementations and beyond. The artifact for CREAL, including all source code and data, is permanently available [73].

6.1 ILLUSTRATIVE EXAMPLES

```
// Input: [*, 75, 48]   => Output: 2
int print_dec(
 char*buf,int max,unsigned value){
  int i = 0;
  if (value == 0) {
    if (max > 0) {
      buf[0] = '0';
      i = 1;
    }
  } else while(value && i < max) {
    buf[i++] = '0' + value % 10;
    value /= 10;
  }
  return i;
}
```

a. A real-world function.

```
1  int a, b;
2  int main() {
3    int c = 1, d = 0;
4    long j[2];
5    for (a = 0; a < 2; a++) {
6      {
7  //Profile: a={0,1},b={0,1},c={1},d={0}
8        b = d + a;
9      }
10     j[b] = 1;
11   }
12   printf("%d\n", j[0]);
13 }
```

b. The seed program.

```
int a, b;

/* A function from FreeBSD.*/
int print_dec(
 char*buf,int max,unsigned value){
  int i = 0;
  if (value == 0) {
    if (max > 0) {
      buf[0] = '0';
     i = 1;
    }
  } else while(value && i < max) {
    buf[i++] = '0' + value % 10;
    value /= 10;
  }
  return i;
}

int main() {
  int c = 1, d = 0;
  long j[2];
  for (a = 0; a < 2; a++) {
    {
      char h[2] = {0, 0};
//replace ``d'' with a function call
      b = print_dec(h,c+74,d+48)-2+a;
    }
    j[b] = 1;
  }
  printf("%d\n", j[0]);
}
```

c. Our generated new program.

Figure 6.1: The program in (c) triggered a latent miscompilation bug in GCC. It is generated by replacing the variable `d` from line 8 in (b) with a carefully designed function call to (a). The synthesized function call is highlighted in gray in (c).

We use a concrete GCC miscompilation bug to illustrate (1) what features in the injected real-world function trigger the compiler bug, (2) why current generators and mutators cannot trigger this bug, and (3) how CREAL injects a real-world function into the seed program. Since Csmith [159] is by far the most powerful and widely used generator for producing generic C programs [17, 139], in this thesis, we use Csmith as the seed program generator. To facilitate our presentation, we only show reduced and simplified programs.

Figure 6.1c shows a program generated by CREAL that triggered a long-latent GCC miscompilation bug. This program is produced by injecting the real-world function `print_dec` shown in Figure 6.1a into the seed program shown in Figure 6.1b.

➤ Q1: What features in the real-world function are essential to trigger the bug?
The function "`print_dec`" is extracted from *the FreeBSD project* [8]. Its definition can be found in Figure 6.1a. This function converts a decimal integer "`value`" into a string and returns its number of digits as `i`. Now, let us focus on the program in Figure 6.1c. The function call "`print_dec(h, c+74, d+48)`" in line 26 should always return 2 because `c+74=75`, which is a 2-digit number. Thus, `b = a`. The correct output from this program should be `1` as `j[0]` is set to 1 in line 28. The executable compiled by GCC at -O3, however, outputs `13368`. The root cause is that when GCC unrolls the "`while`" loop in lines 12-15, the complex control flow leads to incorrect memory partition for arrays `j` and `h`. Consequently, values in `j` are polluted by `h` and become incorrect. The complex `while` loop in "`print_dec`" is essential to trigger the bug — removing any expression such as "`i<max`" (line 12), "`value%10`" (line 13), or "`buf[i++]`" (line 13) from the loop cannot trigger the bug.

➤ Q2: Why current generator- and mutator-produced programs cannot trigger this bug?
As discussed above, the complex "`while`" loop is the key to triggering the bug. However, as has discussed before, generators like Csmith can only generate constant-bounded "`for`" loops such as

```
for (i = 0; i < 2; i++) {          for (i = 0; i != 2; i-=v) {
  ...                                ...
}                                  }
```

Csmith is conservative in generating loops to guarantee termination and avoid undefined behavior. All the loops are bounded by constant values, and loop indexes are only modified in the "increment" parts of `for` loops. The `while` loop from the real-world function in Figure 6.1a, on the other hand, has more features: (1) it has two loop indexes, (2) both loop indexes are modified in the loop body, and (3) the loop is not constant-bounded.

8 https://github.com/freebsd/freebsd-src/blob/main/contrib/unbound/compat/snprintf.c

The Csmith implementation does not support these features. Consequently, mutators relying on generators cannot support these features, either.

➤ Q3: How does our approach generate the new program?
Our approach aims to inject real-world function calls into a seed program. Real-world functions need to be available in the first place. We design a function extractor to extract and preprocess functions from real-world projects. Assume that "`print_dec()`" is the only function we collected. After collection, we generate a driver function to capture valid input/output pairs for the function. One possible driver function for `print_dec()` is

```
int main () {
  char arr[2];
  int ret = print_dec(arr, 75, 48);
  print("%d\n", ret);
}
```

Executing the driver function can tell us that when calling the real-world function with "`print_dec(arr, 75, 48)`" where `arr` is pointing to a sufficiently large memory, the return value of "`print_dec()`" is "`2`". This I/O information will be added along with the static function information into our function database. For example, the first line in Figure 6.1a indicates one input/output pair for the function. We apply extensive analysis such as sanitizers [130] to guarantee that there is no undefined behavior when calling the function with the cached inputs. More details will be included in Section 6.3.2.

The resulting program should be well-formed. To this end, we construct an *equivalence modulo inputs* (EMI) variant when injecting calls, which means the new program has an input/output behavior identical to the seed program. For example, the injected call "`print_dec(h, c+74, d+48)-2`" is always evaluated to `0` at run-time, which is identical to the replaced variable "`d`". This equivalence guarantees that the resulting new program is also well-formed as long as both the seed program and the injected function are well-formed. Specifically, given the seed program in Figure 6.1b, CREAL works as follows:

Step 1. Seed analysis: identify variables or constants after which function calls can be injected. For example, the variables "`d`" and "`a`" from line 8 in Figure 6.1b are valid candidates. Other valid expressions include "0" and "2" in line 5, "`b`" and "1" in line 10, *etc.*

Step 2. Seed profiling: instrument and profile the program to collect values of in-scope variables. Suppose that only variable "`d`" in line

8 is matched. Since "`a,b,c,d`" are in-scope variables in line 8, we collect values of variables `a,b,c,d` in line 8 with our instrumentation. As indicated by line 7, the profile includes all run-time values of these variables: both "`a`" and "`b`" have multiple values, while "`c`" and "`d`" have single values.

Step 3. **Function call synthesis:** synthesize a function call to the real-world function. Since "`d`" is selected, our target is to synthesize a function call that can replace "`d`" while maintaining the execution semantics. We then (i) *Choose a function from the database.* We assume "`print_dec()`" is chosen. (ii) *Synthesize the input parameters to the chosen function.* All input parameters should be identical to the cached input so that we know the return value from the function call during synthesis. Because the first parameter of "`print_dec()`" is a pointer, we place "`char h[2]`" ahead of the current statement as shown in line 24 in Figure 6.1c and use "`h`" as the first parameter. We do not use any pointer from the seed program to avoid modification to the seed program's memory state. Because the second and last inputs are 75 and 48, we can simply use "`print_dec(h,75,48)`" as the function call. In order to build data dependency between the seed program and the injected function, we use in-scope variables as input parameters. For example, since "`c=1`" and "`d=0`", we use "`print_dec(h,c+74,d+48)`" as the function call, which returns the same value as "`print_dec(h,75,48)`". (iii) *Return value compensation:* compensate the return value of the function call to cancel its effect. Since "`print_dec()`" returns `2`, we append "`-2`" after the function call. The final synthesized expression is "`print_dec(h, c+74, d+48)-2`", which is evaluated to `0=d` and thus does not modify the run-time semantics of the seed program.

The final generated program is shown in Figure 6.1c. The injected function call builds dependency between the seed program and the real-world function, bringing the rich features from the real-world function into the seed program. In practice, CREAL will insert more than one function call to enhance the expressiveness of the final program further.

6.2 APPROACH

This section presents our real-world function injection approach. Section 6.2.1 introduces basic concepts about the constructed function database.

Section 6.2.2 describes the high-level algorithmic sketch of our approach. Sections 6.2.3 to 6.2.5 detail the main steps in the sketch. Section 6.2.6 presents our implementation of CREAL.

6.2.1 *Function Database*

Our approach is driven by real-world functions. We design and build a function extractor that can automatically extract, transform, and profile functions from source code files. We defer the technical details of function database construction to Section 6.3. Let us assume a constructed function database $\mathcal{D}_F$ is available. For each function $F_i \in \mathcal{D}_F$, F_i has the following key properties:

1. F_i takes only numeric input types and returns a numeric value.
2. F_i does not contain any other function calls.
3. F_i is pure, meaning that F_i has deterministic outputs and has no side effects.

The first property allows us to avoid passing pointers from the seed program as input parameters, preventing uncontrolled modifications to the seed program's memory state. For the illustrative function "`print_dec()`" in Figure 6.1a that takes a pointer as input, we design a transformation that transforms it into a numeric input-only function. Technical details will be clarified in Section 6.3.1. To simplify our presentation, *we will always assume that all functions take only numeric input types*. The second property indicates that each function can be compiled and executed independently. The last property allows us to fully model a function's semantics by its input/output pairs.

6.2.2 *Algorithmic Sketch*

Our high-level idea is to substitute an expression in a seed program with a function call to a real-world function. The resulting program should have the same semantics as the seed program. Algorithm 9 presents the general process of generating a new program by injecting real-world functions into a seed program $\mathcal{P}$. In general, $\mathcal{P}$ should be deterministic, well-formed (*e.g.*, absence of undefined behaviors), and can be compiled into an executable. Given $\mathcal{P}$ and an input $\mathcal{I}$ to the program, our generator works as follows:

Algorithm 9: Program generation

```
1  procedure Generate(Seed Program $\mathcal{P}$, Input $\mathcal{I}$, Function Database $\mathcal{D}_F$):
       // find all matched expressions $expr$
2      $\mathcal{E} \leftarrow$ GetMatchedExpr($\mathcal{P}$)
       // profiling the program at all program locations of $\mathcal{E}$
3      $\widehat{prof} \leftarrow$ Profile($\mathcal{P}$, $\mathcal{I}$, $\mathcal{E}$)
4      $F\_list \leftarrow [\,]$
5      foreach $expr \in \mathcal{E}$ do
           // decide if this expression needs to be substituted
6          if IsAlive($expr, \widehat{prof}$) and FlipCoin() then
               // select a function from the database
7              $F \leftarrow$ SelectFunction($\mathcal{D}_F$)
               // synthesize the function call
8              $expr' \leftarrow$ SynFuncCall($expr, F, \widehat{prof}$)
               // substitute the original expression with the
                  function call
9              $\mathcal{P} \leftarrow$ InsertFunc($\mathcal{P}, expr, expr', F$)
10             if $F \notin F\_list$ then
                   // remember all used $F$
11                 $F\_list$.append($F$)
       // insert function declarations
12     InsertFuncDecl($\mathcal{P}, F\_list$)
13     return $\mathcal{P}$
```

Step 1. *Expression matching* (line 2): find all expressions in the seed program $\mathcal{P}$ that satisfy a set of predefined criteria (see Section 6.2.3). These expressions will later be replaced with function calls in a certain probability.

Step 2. *Program profiling* (line 3): instrument and run the program $\mathcal{P}$ on input $\mathcal{I}$ to collect the execution profile $\widehat{prof}$, which contains all the needed run-time information at the program location of each matched expression. (see Section 6.2.4)

Step 3. *Skipping an expression* (line 6): for the matched expressions, we do not aggressively substitute all of them with function calls. First, we query the execution profile to see if $expr$ is alive, *i.e.*, exists in the live rather than dead code region. Then, for live expressions, we do substitution in a certain probability. The function `FlipCoin()` returns either `true` or `false`. The probability for `FlipCoin()` to return `true` can be configured in our CREAL implementation. The reason why we skip dead code regions is that previous work [67, 139] has shown that mutating on live regions is more likely to exercise interesting compiler optimizations and trigger bugs.

Step 4. *Function selection and function call synthesis* (lines 7-8): a function $F \in \mathcal{D}_F$ is randomly selected. With the execution profile $\widehat{prof}$, we know the evaluation results of $expr$ and a set of in-scope variables. The `SynFuncCall` procedure will then synthesize a function call expression that is evaluated to the same value as $expr$. (see Section 6.2.5)

Step 5. *Function call insertion* (lines 9-11): the new expression $expr'$ with a function call will replace the original expression $expr$ in $\mathcal{P}$. This function will also be appended into F_list.

Step 6. *Function declaration insertion* (line 12): after the iteration is over, insert the declarations of all used functions to the beginning of $\mathcal{P}$, making it to be compilable and executable.

As indicated in the algorithm, the generation happens iteratively over all matched live expressions. The resulting program contains more than one real-world function call. For the rest of this section, we will introduce in detail the three key steps, *i.e.*, *expression matching*, *program profiling*, and *function call synthesis*.

$$
\begin{aligned}
S &::= E \oplus \langle e \rangle \mid \langle e \rangle \oplus E \mid S' \odot E \\
E &::= \langle c \rangle \mid \langle v \rangle \mid * \langle v \rangle \mid \langle v \rangle [\langle e \rangle] \\
\langle c \rangle &::= \textit{constants} \\
\langle v \rangle &::= \textit{variables} \\
\langle e \rangle &::= \textit{all expressions in C} \\
\oplus &::= \textit{binary operators}, \text{e.g.}, +|-|*|\%, \text{etc.} \\
\odot &::= \textit{assignment operators}, \text{e.g.}, = |+= |*=, \text{etc.} \\
S' &::= \textit{left operand of } \odot \textit{ (ignored)}
\end{aligned}
$$

Source code	Matched expressions
`a + b`	`a, b`
`(a + b) - c`	`a, b, c`
`*a + b`	`*a, b`
`a = b + 1`	`b, 1`
`a += b + c[1]`	`b, c[1]`

Figure 6.2: Syntax rules for matching expressions `E` appearing in `S`. Note that, `E` is of numeric type, such as `int`.

6.2.3 *Expression Matching*

Given a seed program $\mathcal{P}$, we statically scan $\mathcal{P}$ to find all expressions where function calls can be injected. Figure 6.2 gives the syntax rules used for matching target expressions and a set of matched expression examples. There are two main criteria:

- The expression has to be non-left, *i.e.*, not on the left-hand side of an assignment expression. This criterion ensures syntactic validity after we inject a function call. For instance, replacing "`a`" in "`a = b + 1`" with a function call violates the C language specification.

- The expression has a numeric type. Since all functions in the database have numeric input/output types, we match numeric expressions only for type compatibility.

A valid expression can contain multiple matched expressions. For example, "`(a+b)-1`" contains three expressions that satisfy our criteria, namely "`a`", "`b`", and "`1`". Specifically, we match three types of expressions, *i.e.*, (1) constants, (2) variables having numeric types, and (3) pointer dereferences or array accesses having numeric types such as "`*a`" or "`c[1]`" of type `int`.

6.2.4 *Program Profiling*

Program profiling collects the run-time variable values at target program locations. The target program locations $\mathcal{L}$ are a set of lines that contain at least one matched expression. Let us denote all matched expressions at the program location l as $\mathcal{E}_l$ and all in-scope variables at l as $\mathcal{V}_l$. It is clear that *all variables from $\mathcal{E}_l \in \mathcal{V}_l$.* With this notation, we define the execution profile as follows:

Definition 8 (Execution Profile)**.** *Given a program $\mathcal{P}$, an input $\mathcal{I}$, and program locations $\mathcal{L}$. By running $\mathcal{P}$ on the input $\mathcal{I}$, the execution profile $\widehat{prof}$ records the values of expressions in $\mathcal{V}_l$ at each program location $l \in \mathcal{L}$.*

Figure 6.3 shows the execution profile of an example program. In this program, the target program locations are $\mathcal{L} = \{5, 6, 8\}$ because only these three lines contain matched expressions. Note that we do not touch variable definitions, and thus, lines $\{1, 3, 7\}$ are not matched. In line 5, the variable `b` and constant `1` are matched expressions; variables `a`, `b` and `g` are in the variable set $\mathcal{V}_5$ as they are all in-scope. In line 8, there is an extra in-scope variable `c`. The execution profile, as shown in the bottom right of the figure, records all observed run-time values of variables in each $\mathcal{V}_i$.

For a non-trivial seed program $\mathcal{P}$, the number of in-scope variables $\mathcal{V}_i$ at each program location is typically large. Recording all of their values is thus costly. Since only a part of variables from $\mathcal{V}_i$ will be used for synthesis, to reduce the profiling cost, our implementation of Creal randomly selects a small yet sufficient subset of in-scope variables in each $\mathcal{V}_i$. To facilitate easy access to the execution profile $\widehat{prof}$, we design the following queries:

- **`GetStableVariable(`**$\widehat{prof}, l$**`)`**: returns all in-scope variables where *the variable has only one run-time value.* For example, getting the in-scope variables by calling `GetStableVariable(`$\widehat{prof}$`, 5)` returns only {`b, g`} but not `a`. It may also return none if no stable variable is available. In this case, the following synthesis will use constants.

- **`GetVariableValue(`**$\widehat{prof}, l, v$**`)`**: returns the run-time value(s) of variable v at location l, *e.g.*, `GetVariableValue(`$\widehat{prof}$`, 5, b)` returns {`1`}.

6.2.5 *Function Call Synthesis*

Given a selected expression *expr*, we synthesize a new expression *expr′*, such that

$$[\![expr']\!] = [\![expr]\!]$$

where $[\![x]\!]$ denotes the run-time value of expression x. The above equation indicates the run-time values of *expr* and *expr′* are identical. Let $\mathcal{P}[expr/expr']$ be a program formed by replacing *expr* with *expr′* at the same source location. The run-time value equivalence of *expr* and *expr′*

Algorithm 10: Function call expression synthesis.

```
1  procedure SynFuncCall($Program\ Location\ l$, $Profile\ \widehat{prof}$, $Database\ \mathcal{D}_F$,
     $Target\ Value\ \overline{\texttt{val}}$):
       // randomly select a function from the database and get
          its input set
2      $F_i \leftarrow$ SelectFunction($\mathcal{D}_F$)
3      [$\overline{\texttt{inp}_1}$, $\overline{\texttt{inp}_2}$, $\cdots$ , $\overline{\texttt{inp}_m}$ ] $\leftarrow F_i$.input
       // initialize the function call template
4      FC = “ $F_i$.name (<para>$_1$, <para>$_2$, $\cdots$, <para>$_m$)”
       // iteratively synthesize the function parameters
5      foreach $k \in [1 \ldots m]$ do
6          $V \leftarrow$ GetStableVariable($\widehat{prof}$, $l$)
           // sythesize expression $para$ using $V$ such that
              $[\![para]\!] = \overline{\texttt{inp}_k}$
7          $para \leftarrow$ SynthesizeExpression ($V$, $\overline{\texttt{inp}_k}$)
8          FC.Substitute(<para>$_k$, para)
9      $V' \leftarrow$ GetStableVariable($\widehat{prof}$, $l$)
       // synthesize expression $expr'$ using $V'$ and FC such that
          $[\![expr']\!] = \overline{\texttt{val}}$
10     $expr' \leftarrow$ SynthesizeExpression ($V' \cup$ FC, $\overline{\texttt{val}}$)
11     return $expr'$
```

```
 1 int g=4;
 2 int main() {
 3   int a = 0, b = 1;
 4   for (;;) {
 5     a += b + 1;
 6     if (a > g) {
 7       int c = 3;
 8       g = a + c;
 9       break;
10     }
11   }
12 }
```

Program locations: $\mathcal{L} = \{5, 6, 8\}$

Matched expressions:
$\mathcal{E}_5 = \{\texttt{b, 1}\}$
$\mathcal{E}_6 = \{\texttt{a, g}\}$
$\mathcal{E}_8 = \{\texttt{a, c}\}$

Execution profile:
$\mathcal{V}_5 :$ $\texttt{a} = \{0,2,4\}, \texttt{b} = \{1\}, \texttt{g} = \{4\}$
$\mathcal{V}_6 :$ $\texttt{a} = \{2,4\}, \texttt{b} = \{1\}, \texttt{g} = \{4\}$
$\mathcal{V}_8 :$ $\texttt{a} = \{4\}, \texttt{b} = \{1\}, \texttt{c} = \{3\}, \texttt{g} = \{4\}$

Figure 6.3: The program on the left is the seed program. $\mathcal{L}$ is the program locations where we have matched expressions $\mathcal{E}$. The execution profile shows the run-time values of in-scope variables at each program location.

guarantees that (1) $\mathcal{P}[expr/expr']$ has the same output as $\mathcal{P}$, and (2) if both $\mathcal{P}$ and $expr'$ are well-formed, *e.g.*, free of undefined behavior, $\mathcal{P}[expr/expr']$ is also a well-formed program.

There are two cases for *expr*: (1) *expr* is stable, meaning that $[\![expr]\!]$ is a single value $\overline{\texttt{val}}$, *e.g.*, variable "b" at line 5 in Figure 6.3. In this case, we require $[\![expr']\!] = \overline{\texttt{val}}$. (2) *expr* is unstable, meaning that $[\![expr]\!]$ has different values at run-time, *e.g.*, variable "a" at line 5 in Figure 6.3. In this case, we synthesize the new expression as "$expr' = expr + \widehat{expr}$", where $[\![\widehat{expr}]\!] = 0$. This ensures that $[\![expr']\!] = [\![expr]\!]$. The second case can be viewed as the first case by treating $\widehat{expr}$ as the target expression $expr'$. Without loss of generality, our presentation always assumes that *expr* is stable, and our target is to synthesize a new expression $expr'$ that evaluates to $[\![expr]\!] = \overline{\texttt{val}}$.

Algorithm 10 shows the general procedure of synthesizing a new expression. The core merit of this synthesis is to guarantee that the new expression has the same run-time value as the target expression. It first randomly selects a function item F_i from the constructed function database $\mathcal{D}_F$ and obtains the function input set (lines 2-3). Note that all $\overline{\texttt{x}}$ such as $\overline{\texttt{inp}_1}$ and $\overline{\texttt{val}}$ are concrete numeric values, *e.g.*, "0", "2", *etc.* Then, it initializes a function call template as FC (line 4). This template is of string format and has a set of placeholders for input arguments, *i.e.*, $\texttt{<para>}_1$, $\texttt{<para>}_2$, $\cdots$, and $\texttt{<para>}_m$. Next, it tries to replace all these placeholders with concrete expressions. For each of the parameter placeholder $\texttt{para}_k$, it obtains stable

$$\begin{aligned}
\langle e \rangle &::= \langle c \rangle | \langle v \rangle | (\langle e \rangle) | \ominus \langle e \rangle \\
&\quad | \langle e \rangle \oplus \langle e \rangle | \langle F \rangle \\
\langle c \rangle &::= \textit{constants} \\
\langle v \rangle &::= \textit{variables} \\
\langle F \rangle &::= \textit{function call} \\
\ominus &::= \textit{unary operators, e.g.,}\ ! | \sim \\
\oplus &::= \textit{binary operators, e.g.,}\ + | * | \%
\end{aligned}$$

Figure 6.4: BNF grammar for expression synthesis.

variables V from the execution profile (line 6). With the grammar shown in Figure 6.4, it uses V and synthesizes an expression *para* that evaluates to $\overline{\texttt{inp}_k}$ (line 7). The parameter placeholder `<para>`$_k$ is then substituted by the synthesized parameter expression *para* (line 8). After the for loop, the function call to "F_i.`name`(...)" is evaluated to F_i.`output`. Finally, with the grammar shown in Figure 6.4, it synthesizes an expression with a set of stable variables V' and the synthesized function call `FC`. This new expression $expr'$ is now evaluated to the target value $\overline{\texttt{val}}$.

Example. We use an example to illustrate the algorithm. Let us assume the target value is $\overline{\texttt{val}} = 5$:

- In line 2, suppose the selected function is "`int real(int)`", which takes an integer as input and returns an integer value.
- In line 3, suppose the single input is $\overline{\texttt{inp}_1} = 2$ and the function return is 1.
- In line 4, the template is then "`real(<para>`$_1$`)`".
- In lines 6-7, suppose the stable variables V is $\{ b \}$ and "b" has a single value 1. One instance of the synthesized expression can be "$b + 1$", which evaluates to $2 = \overline{\texttt{inp}_1}$.
- In line 8, the synthesized input expression is then "$b + 1$", and the function call `FC` is "`real(`$b + 1$`)`".

- In line 10, suppose the stable variables V' is $\{ g \}$ and "g" has single value 4. One instance of the synthesized expression can be "$(-\texttt{real(}b+1\texttt{)} * 2) + g * 2 - 1$" because

$$\begin{aligned}(-\texttt{real(}b+1\texttt{)} * 2) + g * 2 - 1 &= (-1 * 2) + g * 2 - 1 \\ &= (-1 * 2) + 4 * 2 - 1 \\ &= 5 = \overline{\texttt{val}}\end{aligned}$$

The synthesized expression $expr'$ fuses the function call with variables in the seed program. This fusion builds rich dependencies between the seed program and the inserted functions. The original expression $expr$ is then replaced by $expr'$.

6.2.6 *Implementation*

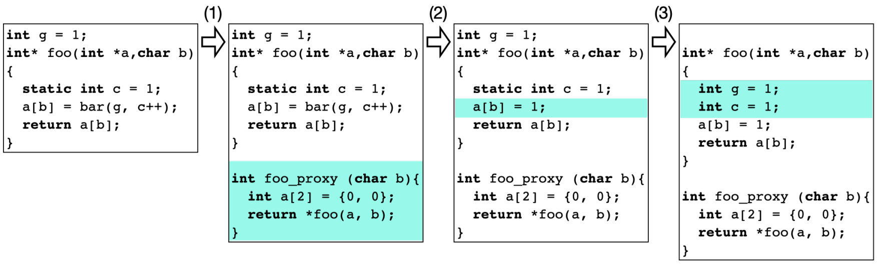
We implemented the proposed program generator in CREAL. Taking a seed program $\mathcal{P}$ and a function database $\mathcal{D}_F$ as inputs, CREAL utilizes Clang's Libtooling [81] to instrument profiling code into $\mathcal{P}$ and collects its execution profile. When generating a mutant, CREAL replaces multiple expressions in the seed program with different synthesized function call expressions.

Type conversion. In the above presentation, we assumed that there is only one numeric type in the program. In practice, the C language supports many types, such as `int`, `char`, and `unsigned`. When an operation happens between values of different types, the compiler will do implicit type conversions. For example, the evaluation results of "$a + b$" are different when "a" has different types:

$$\begin{aligned}\texttt{int}\ a = 0; \texttt{int}\ b = -1; \quad &\Rightarrow \quad a + b = -1\ \texttt{(int)} \\ \texttt{unsigned}\ a = 0; \texttt{int}\ b = -1; \quad &\Rightarrow \quad a + b = 4294967295\ \texttt{(unsigned)}\end{aligned}$$

For the second case, b is cast into `unsigned`, and thus, its value becomes 4294967295 (the maximum value for a 32-bit unsigned integer). In CREAL, we carefully evaluate the expression values when type conversion is possible. We also add explicit casts to make sure the type of $expr'$ is the same as the original expression $expr$.

Avoiding undefined behavior. When synthesizing expressions in CREAL, it is important to guarantee that the synthesized expressions are well-formed. Since our expressions involve only arithmetic operations, we only need to



```
int g = 1;
int* foo(int *a,char b)
{
  static int c = 1;
  a[b] = bar(g, c++);
  return a[b];
}
```

```
int g = 1;
int* foo(int *a,char b)
{
  static int c = 1;
  a[b] = bar(g, c++);
  return a[b];
}

int foo_proxy (char b){
  int a[2] = {0, 0};
  return *foo(a, b);
}
```

```
int g = 1;
int* foo(int *a,char b)
{
  static int c = 1;
  a[b] = 1;
  return a[b];
}

int foo_proxy (char b){
  int a[2] = {0, 0};
  return *foo(a, b);
}
```

```
int* foo(int *a,char b)
{
  int g = 1;
  int c = 1;
  a[b] = 1;
  return a[b];
}

int foo_proxy (char b){
  int a[2] = {0, 0};
  return *foo(a, b);
}
```

Figure 6.5: Step-by-step transform an invalid function into a pure function.

avoid integer overflow. This can be easily achieved because we know the concrete evaluation values of all involved variables.

Generated programs. Our approach is general and can, in principle, take programs from any generators as input as long as they are deterministic, well-formed, and can be compiled into an executable. In our implementation, we use Csmith as the default generator to generate seed programs. In this case, every seed program is in a single source file. After injecting function calls, CREAL includes function definitions for all used functions in the source file. The generated program by CREAL is thus a single source file, as exemplified in Figure 6.1c.

6.3 CONSTRUCTING FUNCTION DATABASE

This section presents details about constructing a function database to fuel our approach.

6.3.1 *Extracting and Transforming Functions*

We design a function extractor to obtain functions from source code files. We utilize Clang's Libtooling [81] for implementation. The extractor finds all function definitions in a pre-processed input program, *e.g.*, programs processed by "`clang -E`". The principal selection criteria are (a) the function only involves primitive types, and (b) the function should be pure, *i.e.*, with deterministic outputs and no side effects. The first requirement allows us to compile and execute functions without dealing with user-defined types. The second requirement allows us to model the execution of a function purely by its I/O behavior. In the context of C, this means

1. *The function does not modify its arguments.* Some functions take pointers as arguments, the memory locations pointed to by which can be modified after invoking the function. Instead of discarding such functions, we synthesize proxy functions for them to avoid possible modifications to arguments. For example, Figure 6.5 (1) shows how we use a proxy function `foo_proxy` to hide the pointer parameter in the original function `foo`. Note that the proxy function also dereferences the return pointer to make it also a primitive type.

2. *The function does not call other functions that have side effects.* Real-world functions may call other functions. Analyzing other functions to avoid side effects is theoretically feasible but would complicate our implementation and make it fragile. Instead, we choose to remove all other function calls from a function definition. To maintain syntactic correctness, we replace each call with a randomly chosen yet type-compatible value. For instance, Figure 6.5(2) shows that the extractor replaces the function call "`bar(g, c++)`" with an integer value "`1`".

3. *The function does not modify global or static variables.* The value of global or static variables may be different when calling the same function multiple times. Such side effects break our intention of modeling a function by its I/O behaviors. Our extractor will remove all "static" keywords from a function definition and then lower all global variables into local variables. For example, the extractor changes variables `g` and `c` as shown in Figure 6.5(3).

Since we utilize a function's I/O for program generation, we also exclude functions that take no inputs or return void. With the designed function extractor, we can extract a set of functions from real-world projects. We denote the constructed function database as $\mathcal{D}_F = \{F_1, F_2, \cdots, F_n\}$, where each F_i contains the static information regarding the function and has the following information:

- F_i.`name`: the unique name of the function in the database, *e.g.*, "`foo_proxy`" in Figure 6.5.

- F_i.`arg_type`: the types of each argument, *e.g.*, ["`char`"].

- F_i.`ret_type`: the return type, *e.g.*, "`int`".

- F_i.`def`: the function definition, *e.g.*, "`int* foo(...){...} int foo_proxy(...){...}`".

```
1  #include <stdio.h>
2
3  /*Placeholder for function definition*/
4  F.def;
5
6
7  int main() {
8    /* randomly select values */
9    F.arg_type[0]  p1 = RAND();
10   F.arg_type[1]  p2 = RAND();
11   ...
12   /* invoke the function */
13   F.ret_type ret = F.name(p1,p2,...);
14
15   /* output the IO */
16   LOG(p1); LOG(p2); ...;
17   LOG(ret);
18 }
```

a. Driver template.

```
1  #include <stdio.h>
2  /* function definition */
3  int add(char a, int b) {
4     int r = a + b;
5     return r;
6  }
7  int main() {
8    /* randomly select values*/
9    char   p1 = RAND(char);
10   int    p2 = RAND(int);
11
12   /* invoke the function */
13   int   ret = add(p1, p2);
14
15   /* output the IO */
16   LOG(p1); LOG(p2);
17   LOG(ret);
18 }
```

b. The driver function for "add"

Figure 6.6: Template for driver functions and an example driver function.

6.3.2 *Constructing Function Database with I/O*

Caching the input/output (I/O) behaviors of a function can help us precisely synthesize a function call and avoid ill-formed functions prior to synthesis. For each function in $\mathcal{D}_F$, we utilize a driver function to learn (1) its I/O pairs, and (2) whether or not the function is well-formed, *i.e.*, whether it contains undefined behavior. Figure 6.6a shows the template for the driver function, and Figure 6.6b shows an example driver function for the function "`add()`". For a function $F_i \in \mathcal{D}_F$, the general I/O construction works as follows:

1. Include the function definition F_i`.def` in the driver template, *i.e.*, line 4 in Figure 6.6a and lines 3-6 in Figure 6.6b.
2. For each argument of F_i, synthesize a statement to randomly pick a value with this type, *i.e.*, lines 9-11 in Figure 6.6a and lines 9-10 in Figure 6.6b. "`RAND()`" function only returns values within the valid range for the given type. For instance, `RAND("char")` returns values between -128 and 127.

3. Synthesize a function call statement and put the return value into a variable with type F.`ret_type`, *i.e.*, line 13 in Figure 6.6a and line 13 in Figure 6.6b.

4. Log the inputs and output, *i.e.*, lines 16-17 in Figure 6.6a and lines 16-17 in Figure 6.6b.

The synthesized driver function is then compiled and executed. We use sanitizers [130] and CompCert [69, 70] to examine whether or not the driver function is well-formed. If all examinations are successful, we save the logged input/output pair into the database. For instance, one valid input/output pair of Fig. 6.6b is {`input=[1,2]`, `output=3`}. In practice, we will try to run each driver function multiple times to save more than one input/output pair into the database. Now, new items F_i.`input` and F_i.`output` are added to the database. *At this point, all functions in the database are free of undefined behaviors when calling them with the inputs cached in* F_i.`input`.[9]

6.4 EVALUATION

This section presents the details of our extensive evaluation of CREAL, demonstrating its practical effectiveness. In a nine-month period until mid-November 2023, we ran CREAL to test two open-source compilers, GCC and LLVM. Our testing results are summarized as follows:

- CREAL *has found many new bugs.* Within nine months of testing, CREAL has detected 132 bugs, of which compiler developers have confirmed 121 bugs and fixed 97 bugs.

- CREAL *has found many miscompilation bugs.* Of all 132 detected bugs, 79 (60%) of them are miscompilation bugs, the most severe and hard-to-detect compiler bug type.

- CREAL *has found many latent bugs.* We reported 41 latent bugs in GCC and LLVM. These bugs have escaped all previous generation- and mutation-based compiler testing techniques.

9 This guarantee relies on the robustness of sanitizers and CompCert. We did not meet any issues in practice, although there are reports complaining sanitizers of missed cases [61] including our UBFUZZ [78].

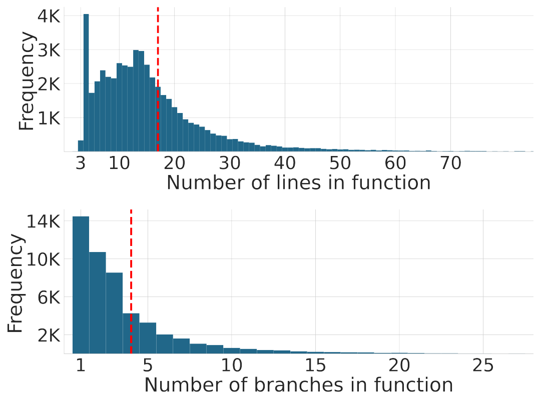


Figure 6.7: Statistics about #lines and #branches of functions in the constructed database.

6.4.1 *Experimental Setup*

Compiler versions. Our evaluation focuses on widely used and mature production C compilers, GCC and LLVM. To avoid reporting previous known bugs, we used CREAL to test the latest development versions of both GCC (from `80d6f89` to `5476de2`) and LLVM (from `34ae308` to `0289dad`). We used all five standard optimization levels, *i.e.*, `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`, in both compilers for extensive testing.

Seed programs. We used Csmith [159], a mature random C program generator, to generate seed programs for CREAL. Csmith has been widely adopted by many compiler testing techniques [17, 31, 82] and has been *de facto* default program generator for C compiler testing. There are three main strengths in using Csmith: (1) it can generate complex programs, providing CREAL rich opportunities to mutate on; (2) all Csmith-generated programs are self-contained, meaning that they do not take external inputs and can be executed; and (3) Csmith-generated programs and their mutants can be effectively reduced with existing tools like C-Reduce [124].

Function database. We used our function extractor to extract functions from open-source projects. AnghaBench [27] provides intermediate access to the source files of 146 most starred open-source C projects on GitHub, such as OpenSSL, PHP, and Linux kernel, to name a few. On top of AnghaBench, we crawled over one million functions from these 146 projects. After extraction and I/O generation as described in Section 6.3, we collected a function database of ***51,356 functions***. The majority of crawled functions are discarded due to unsupported input or return types. We count the number of lines and branches in our collected functions. Figure 6.7 shows the distributions. We can see that most functions have fewer than 30 lines and 10 branches. On average, the number of lines in a function is 17, while the number of branches in a function is 4.

Hardware. We conducted all our evaluations on two Linux servers running Ubuntu 20.04 LTS. Both are equipped with an AMD EPYC 7742 64-core CPU and 256GB RAM.

Testing process. Our testing process runs continuously and is fully automated. We first use Csmith to generate a seed program[10], then apply CREAL to generate 10 mutants. The `FlipCoin()` probability in Section 6.2.2 is set to 20%. Then, we use both GCC and LLVM with different optimization levels to compile and run these programs. Since all mutants are semantics-preserving, we use the output from the seed program as the reference oracle. Once a compiler crash or miscompilation is observed, we use C-Reduce to reduce the bug-triggering program into a small reproducing program and then make a bug report with it.

6.4.2 *Quantitative Results: Bug-Finding*

Number of bugs. Table 6.1 summarizes the status of all found bugs in GCC and LLVM. As of mid-November 2023, we have filed a total of 132 bug reports to GCC and LLVM. The compiler developers have acknowledged, *i.e.*, confirmed or fixed, 92% of them (121/132) as previously unknown and new bugs, while they have fixed 73% of them (97/132). In particular, GCC and LLVM developers confirmed or fixed respectively 86% and 96% of our filed bugs. As mature and production compilers, both GCC and LLVM have been extensively tested in industry and academia. The significant number of new bugs highlights the substantial bug-finding ability of CREAL. The "Reported" bugs are still waiting for developers' confirmation. Since there

10 We used the parameters "`-no-volatiles -no-volatile-pointers -no-unions -ccomp`" when invoking Csmith.

Table 6.1: Status of the reported bugs.

Status	GCC	LLVM	Total
Reported	3	1	4
Confirmed	13	11	24
Fixed	38	59	97
Duplicate	5	2	7
Total	59	73	132

Table 6.2: Type of found bugs.

Symptom	GCC	LLVM	Total
Crash	22	31	53
Miscompilation	37	42	79

are other compiler developers, users, and testers reporting bugs, 5 bugs in GCC and 2 bugs in LLVM are marked as "Duplicate".

Types of bugs. Table 6.2 summarizes the types of found bugs. All bugs can be categorized into the following two types: (1) "Miscompilation": The compiler incorrectly compiles the program and produces a wrong executable code, which has a different semantics from the source program, and (2) "Crash": The compiler crashes when compiling the program due to either run-time failures or assertion failures. As the table shows, more than half of the bugs (79 out of 132) are miscompilations, the most critical and hard-to-detect bugs [17, 139].

Importance of bugs. The fix of 97 out of 132 bugs, predominantly miscompilations, has highlighted the critical impact of the bugs. For GCC bugs, developers will classify bugs according to their priority and severity. P3 is the default while P1 is the highest priority. *Developers have to fix all P1 bugs before making the next release.* Out of the 59 GCC bugs, 19 (32%) of them are assigned as P1.

To understand the impact of found bugs, we ran each bug-triggering test case on stable compiler versions since GCC-5 and LLVM-9. Figure 6.8 shows for each compiler version, how many bugs affect it. It indicates that CREAL can find many *long-latent* bugs. Specifically, there are 14 bugs affecting GCC versions earlier than GCC-11 and 7 bugs affecting LLVM versions earlier than LLVM-13. These compiler versions have been released for 2 ∼ 8 years. Since these long-latent bugs have escaped all existing testing techniques, it further confirms the significant bug-finding capability of CREAL.

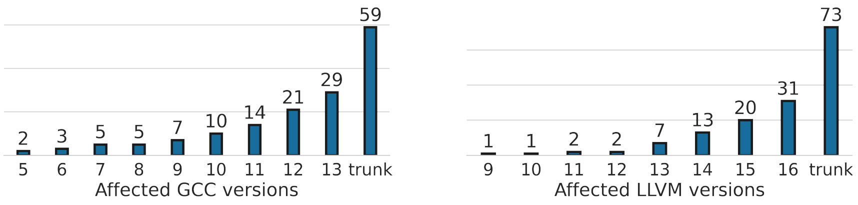


Figure 6.8: Stable compiler versions that are affected by our reported bugs.

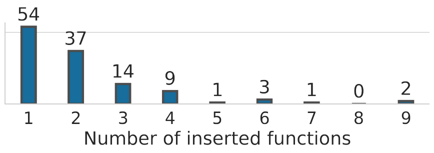


Figure 6.9: Number of inserted functions in reduced bug-triggering programs.

Affected compiler components. We studied all fixed bugs on which compiler developers provided detailed diagnoses and fix messages, allowing us to identify the affected compiler components accurately. Tables 6.3 and 6.4 list all the affected compiler components in LLVM and GCC. One may be concerned that Creal can only find control-flow-related bugs as we only inject function calls into seed programs. However, from the tables, it is evident that these bugs impact a diverse range of compiler components. In both GCC and LLVM, loop transformations and peephole optimizations account for more bugs than other components. This finding is aligned with the existing empirical study [168] that instruction combination (peephole optimization) is one of the most buggy optimizations, and loop optimizations are more bug-prone than other optimizations.

6.4.3 *Bug Characteristics*

For all the unique and new bugs (121 bugs), we identified what real-world functions account for triggering the bug. Because Creal injected multiple functions into a seed program, we automatically removed all irrelevant functions and only kept the functions that were essential to trigger the underlying bug. Note that it is possible that more than one function is

Table 6.3: Affected LLVM components.

Component	#Bugs
Peephole Optimizations	17
Loop Transformations	11
Backend	6
Induction Variable Trans.	6
Scalar Evolution Analysis	5
Global Value Numbering	4
SLP Vectorization	4
Selection DAG	3
Alias Analysis	2
Induction Variable Analysis	2
CFG Transformations	1
Dead Store Elimination	1
Dominance Optimizations	1
Escape Analysis	1
Pass Management	1
Vectorization Optimizations	1

Table 6.4: Affected GCC components.

Component	#Bugs
Loop Transformations	10
Peephole Optimizations	9
CFG Transformations	7
Loop Analysis	3
Value Range Analysis	3
Vectorization	3
Constant Propagation	2
IR Data Structures	2
Dead Store Elimination	1
Jump Threading	1
Liveness Analysis	1
Predictive Commoning	1
Redundancy Elimination	1

required to trigger a bug. With this information, we answer the following questions:

➤ **How many functions are in bug-triggering programs?** Figure 6.9 summarizes the number of real-world functions in each reduced bug-triggering program. The first bar shows that out of all 121 bug-triggering programs, 54 of them have only one single real-world function inserted, demonstrating the ability of one real-world function to enhance seed programs' features. The rest half, nearly all of them, have ≤ 4 inserted functions, with only a few having more than 5 functions. This means that the combination of a small number of functions can further enhance the expressiveness of seed programs.

➤ **Can real-world functions alone trigger bugs?** Figure 6.9 shows that 54 out of 121 bug-triggering programs contain only one real-world function. One may wonder if the seed programs are necessary, *i.e.*, if we can find many compiler bugs without using Csmith-generated seed programs. To answer this question, we used the driver function from Section 6.3 as the seed to build a well-formed program for each function in the database. Then, we compiled and ran these programs with different compilers to test if we could find compiler bugs. *The result is that we did not find any compiler bugs.* This emphasizes the importance of incorporating complex run-time semantics from seed programs.

We also checked if there are functions that lead to multiple bugs. We identified that 5 real-world functions contribute to more than one bug, highlighting the importance of injecting real-world functions into seed programs: *different injections may uncover different bugs.*

The main reason why real-world functions alone cannot trigger bugs is that triggering compiler bugs usually requires specific run-time environments. For example, although the real-world function in Figure 6.1c is essential for triggering the miscompilation bug, the `for` loop and the use of global variables "`a`" and "`b`" in the `main` function provide necessary run-time environments and are also necessary. The driver functions from Section 6.3 can only provide functional but trivial environments and thus are not suitable for finding compiler bugs. Seed programs from random generators like Csmith, on the other hand, can provide the required complex and diverse environments.

Note that Creal can be applied beyond Csmith. The core merit of Creal delivers is that we can significantly boost a generator's expressiveness by fusing real-world code. In general, any program generator that can produce closed and executable programs with non-trivial syntax, such as loops, global variables, and complex data structures, can benefit from Creal.

➤ **What are the unique features in bug-triggering functions?** We manually analyzed each minimized bug-triggering program to figure out what unique code features are in the real-world functions. We find two notable features that appear in 21 bugs:

- *Unbounded complex loop*, where the loop conditions are not bounded by a constant value and have a complex dependency with the loop body. This feature is present in 11 bug-triggering programs. As discussed before, generating complex loops is difficult for random program generators due to the need to guarantee termination and avoid undefined behavior.

Table 6.5: Line coverage (LC), function coverage (FC), and branch coverage (BC) of GCC and LLVM.

Compiler	Generator	LC	FC	BC
GCC	Seeds (1,000)	30.9%	34.3%	19.2%
	Csmith (10,000)	33.6% (+23,897)	35.5% (+1,055)	21.4% (+24,055)
	Creal	**37.2% (+55,761)**	**37.5% (+2,813)**	**24.0% (+52,485)**
LLVM	Seeds (1,000)	33.3%	24.9%	18.2%
	Csmith (10,000)	34.7% (+22,788)	25.8% (+801)	20.3% (+14,166)
	Creal	**35.9% (+42,952)**	**26.9% (+1,864)**	**22.1% (+27,382)**

- *Employing switch/case statements*, where the function employs specific conditional statements "`switch...case...`" to build control flow. This feature is present in 10 bug-triggering programs. Each `switch` often comes with multiple (5 $\sim$ 21) `cases` conditioning on the same expression. Csmith does not yet support this feature.

The remaining bug-related functions have various features. They typically have either unique syntax, such as the above two constructs, or unusual real-world semantics, such as *flipping bits* and *rotating arrays*. Although random program generators support the syntax in these functions, the underlying semantics are nearly impossible to be from random generation. We will include more sample bugs in Section 6.4.7 to demonstrate this point further.

6.4.4 *Code Coverage and Generation Speed*

We conduct code coverage analysis to understand if Creal can increase code coverage. We randomly generated 1,000 seed programs with Csmith. Then, we use Creal to generate 10 mutants per seed, which leads to a total of 10,000 programs from Creal. For a fair comparison against Csmith, we also used Csmith to generate 10,000 random programs. For each set of programs, we collected their function, line, and branch coverage on GCC and LLVM[11]. Table 6.5 reports the results. It is clear that Creal achieves significantly higher code coverage. Compared to the seed programs, Creal remarkably increased line coverage in GCC by 6.3% (55,761 more lines) and in LLVM by 2.6% (42,952 more lines). Creal also shows significantly higher coverage compared to the same amount of Csmith-generated programs. Overall, the large amount of extra code regions signifies that Creal-generated programs cover more code features.

With the same set of seeds, we measured the generation speed of Creal. Generating 10,000 mutants took 8,710 seconds, an average of 0.87 seconds per mutant. During our testing process, we observed that compiling and executing programs consumed most of the time (≥ 3 seconds per mutant), while program generation was not a bottleneck.

6.4.5 *Significance of Function Database*

To evaluate the impact of real-world functions, we construct three variants of Creal:

- Creal-$\frac{1}{2}$: We halve the size of the real-world function database used in Creal. We randomly selected half of the functions and let Creal use this smaller database during generation.
- Creal-$\frac{1}{4}$: We further halve again the size of the real-world function database used in Creal.
- Creal-$_{Csmith}$: Instead of using real-world functions, we extract functions from Csmith-generated programs. We build a new function database of the same size as in Creal, *i.e.*, 50,000 functions from Csmith-generated programs. Then, we set up Creal to use this new function database.

Comparing against Creal-$\frac{1}{2}$ and Creal-$\frac{1}{4}$ helps us understand if more real-world functions in the database can offer richer code features. Comparing against Creal-$_{Csmith}$ helps us understand the necessity and importance of using functions from real-world projects.

Code coverage analysis. We ran these variants on the same 1,000 seeds as in Section 6.4.4. We let each tool generate 10 mutants per seed. Figure 6.10 shows the covered lines by each variant. Overall, our default Creal achieves

11 We used `gcov` for GCC coverage collection and `llvm-cov` for LLVM coverage collection.

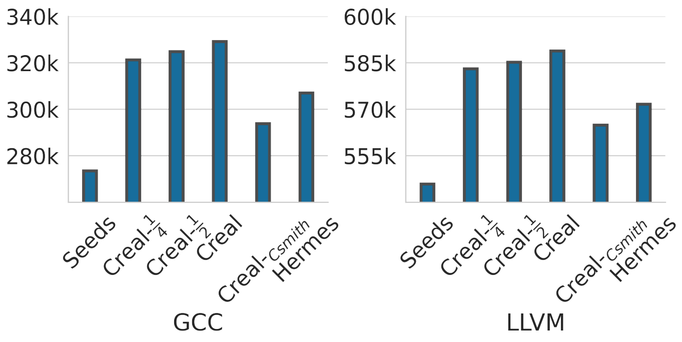


Figure 6.10: Line coverage of different variants of Creal and Hermes.

the highest line coverage in both GCC and LLVM. Compared to both Creal-$\frac{1}{4}$ and Creal-$\frac{1}{2}$, we can see that with the increase of database size, the line coverage also increases. This confirms the necessity of using a large set of real-world functions. Compared to Creal-$_{Csmith}$, which has the same number of functions as Creal, all variants based on real-world functions significantly surpass it. We can further confirm that real-world functions enhance the functionality and expressiveness of seed programs.

Bug-finding analysis. Our analysis of bug-triggering programs in Section 6.4.3 has shown that *nearly all of the bugs are triggered by different real-world functions* except for 5 functions. When using a smaller database such as Creal-$\frac{1}{4}$, it is very likely to exclude many bug-relevant functions. Using such a small database would not be able to detect as many bugs as the full-size database. It is thus clear that *reducing the number of real-world functions hinder the bug-finding capability of* Creal.

We also evaluated the bug-finding capability of Creal-$_{Csmith}$. To this end, (1) we ran Creal-$_{Csmith}$ for one week on the same machine to see if it could find any new bugs, and the result shows that Creal-$_{Csmith}$ failed to find any new bugs. (2) we selected all fixed bugs (97 bugs) found by Creal, then applied Creal-$_{Csmith}$ to mutate each seed program that was used by Creal to trigger the bug. Considering the stochastic nature of Creal, we let Creal-$_{Csmith}$ mutate for one hour, generating more than 200 mutants per seed (instead of 10 used in Creal). In the end, Creal-$_{Csmith}$ re-discovered 3 bugs in LLVM and 0 in GCC, which are all recent regressions. All three bugs are crash bugs, and no miscompilation was found.

6.4.6 *Comparison of* Creal *v.s. Hermes*

We also compared Creal against Hermes [139], which is the most recent and powerful EMI-based compiler testing tool.

Code coverage analysis. We used the same 1,000 seeds as in Section 6.4.4 and applied Hermes to generate 10 mutants per seed. In the end, we obtained 10,000 Hermes-generated programs, the same number as Creal. The last bars in Figure 6.10 show the achieved line coverage of Hermes on GCC and LLVM. Hermes improved line coverage compared to the seed programs, but the improvement is not as significant as Creal. Since all Hermes-injected code snippets are derived from a relatively simple grammar, it is expected to exercise much fewer code features than Creal.

Bug-finding analysis. According to EMI's project website [166], Hermes has been constantly used for finding compiler bugs until now: "*We have maintained our continuous, extensive effort in stress-testing GCC and LLVM to benefit the entire community*". Despite their efforts, the 121 new compiler bugs demonstrate at least the complementary bug-finding capability of Creal. To understand if Hermes can re-discover the bugs found by Creal, as in Section 6.4.5, we selected all fixed bugs (97 bugs) and applied Hermes to mutate each seed program that was used by Creal to trigger the bug. For each seed, we ran Hermes for one hour, generating more than 100 mutants per seed (instead of 10 used in Creal). The result shows that Hermes was able to re-discover 3 bugs.

Despite the evaluation favoring Creal, we stand by the fact that the strengths of both Creal and Hermes are complementary. Similar to Csmith, Hermes can be used as a program generator, on top of which Creal can bring in real-world code features.

6.4.7 *Case Study*

Figure 6.11a: This program triggers a miscompilation bug in GCC. The function `real` is extracted from `FreeBSD` [37], which converts an input numeric variable `dest` into a pointer and then operates over this pointer. GCC incorrectly handles the `while` loop by asserting that this loop must be executed at least once. The reason is that `*ptr` and `len` are from the same variable `j`, and the pointer `*ptr` is used in line 7, which leads GCC to conclude that `len` must not be zero otherwise `*ptr` would deference a null pointer. Line 5, where an integer is converted to a pointer, is essential to trigger this bug. Such corner functionality cannot be covered by generators.

```c
int a, f, i, h;
static int *e, *g;

long real(long dest,int val,long len){
    char *ptr = (char*)dest;
    while (len-- > 0)
        *ptr++ = val;
    return dest;
}

int main() {
    unsigned j = i, c = 0;
    for (a=1; a<2; a++) {
        j = real(j, c, j);
        if (e) break;
    }
    const int **k = &e;
    j && (*g)--; *k = 0;
    printf("%d\n", f);
}
```

a. GCC at -O1 miscompiles this code. The compiled binary produces 2 instead of 1.

```c
static int a=0, *b, *c=&a;  int d,e;
int real(
  int effectnum,int *converted_num){
  switch (effectnum) {
    case 0x0:
      *converted_num = effectnum;
      return 1;
    case 0x4: return 1;
    case 0x5:
      *converted_num = effectnum+1;
      return 0;
    default: return 0; }
}
int main() {
    unsigned h=0;
    for (; a + h <= 6; h = h + 3)
        for (; d; d++);
    int j, *k = c, d = &k != &b;
    int g = real(h, &j);   e = g;
}
```

b. Clang -O1 crashes on this code. It triggers an assertion failure in `SimplifyCFG` component.

```c
int a, b, c, d;
int *e = &c;
unsigned real(unsigned char x) {
  x = ((x>>1)&0x55) | ((x<<1)&0xaa);
  x = ((x>>2)&0x33) | ((x<<2)&0xcc);
  x = ((x>>4)&0x0f) | ((x<<4)&0xf0);
  return x;
}
void h(unsigned g) {
    *e = 8 > real(g + 86) - 86;
}
int main() {
  d = a && b;
  h(d + 4);
  printf("%d\n", c);
}
```

c. GCC at -O1/2/3/s miscompiled the code. The compiled binaries output 0 instead of 1.

```c
static int a = -3, b;
static char c;
int d;
int real(int low, int high) {
    if (low - high < 0x10000L)
        return low;
    return low + (1 % (- low));
}
int main() {
    int *h[] = {&a, &a};
    for (; c <= 8; ++c) {
        int *i = &b;
        *i |= real(a, 8) + d;
    }
    printf("%d\n", b);
}
```

d. Clang at -Os miscompiles this code. The compiled binary produces -1 instead of -3.

Figure 6.11: Sample reduced programs that trigger compiler bugs.

Figure 6.11b: This program triggers an assertion failure at Clang -O1: *"TableSize >= Values.size() && "Can't fit values in table!""*. The function `real` is extracted from `GBStudio` [138] and has more than 20 `case` branches. We omit the majority of them for easier presentation. The compiler incorrectly infers the value range of `effectnum` when simplifying the control flow graph. The multiple branches conditioning on the same variable is important to trigger the bug.

Figure 6.11c: This program triggers a latent miscompilation bug since GCC-10. The function `real` is extracted from `QMKFirmware` [36], which rotates the bit stream of the input `x`. GCC has built-in rotation optimizations, which incorrectly handle this case. The rotation functionality, critical to triggering this bug, is almost impossible for a random program generator to generate.

Figure 6.11d: This program triggers a miscompilation bug in LLVM. The root cause is in the backend, which incorrectly rewrites the expression at line 7. The operand "`low`" and the last "`-low`" are essential to trigger this bug. The function `real` is extracted from `FreeType` [38]. Although it seems that such a function has trivial syntax, the semantics that the same operand "`low`" has to appear twice with negated values is non-trivial.

6.4.8 *Discussion*

Limitations and Extensions. We position Creal as a complementary approach for existing generators. It significantly boosts a program generator's expressiveness by injecting real-world code. Since Creal relies on the function database, it will eventually saturate until new functions or extensions are developed. In our prototype, we only support functions that are pure and have numeric input/output types, which simplified our implementation. In general, as shown in Section 6.4.6, increasing the size of the function database can help find more bugs. Lifting function restrictions will increase the database's size and thus improve the bug-finding capability. However, lifting these restrictions may require some engineering efforts. One applicable way to support non-numeric inputs is to wrap functions with proxy code. For example, we synthesized a proxy function in Figure 6.5 to transform the pointer-based arguments into numeric types. Such a strategy can also be applied to functions with other argument types. However, it is challenging to support functions with unusual and complex features, such as using user-defined `struct/union` types. We consider this as an exciting and orthogonal future work.

Improving existing generators. Bugs found by Creal reveal certain limitations of existing generators and provide possible improvement directions. For example, Figure 6.11b shows that the complex conditions from `switch/case` can trigger compiler bugs. Generators like Csmith, however, do not support this feature. This indicates that it would be beneficial for generators to support this feature for better bug-finding capability. Engineering existing generators to support more features is, in principle, possible but may require significant efforts.

6.5 RELATED WORK

In this section, we discuss related work in the context of compiler testing.

Generative compiler testing. Generative compiler testing utilizes random programs produced by a well-engineered generator to test compilers. Csmith [159] is by far the most impactful program generator for finding C/C++ compiler bugs. It can produce a large number of well-formed programs, and hundreds of compiler optimization bugs were found when it was launched more than ten years ago. Csmith has also been adapted for other compiler testing scenarios. For example, CLsmith [82] modifies Csmith to test OpenCL compilers. YARPGen [85] and its follow-up second version YARPGen v2 [86] are more recent re-designed program generators targeting scalar and loop optimizations, respectively. Together, they have discovered hundreds of optimization bugs in a few years.

A key challenge for generative tools is to guarantee the validity of the generated programs. Extensive engineering efforts are an absolute necessity to support a wide range of yet incomplete language features. Due to the stochastic nature of random generation, certain code semantics such as the ones discussed in Section 6.4.7 have low probabilities of being generated. Creal tackles this challenge by embracing the power of rich open-source projects and requires relatively low engineering efforts. Conceptually, our solution can be effortlessly adapted to any generator-based approach.

Mutation-based compiler testing. Another line of compiler testing is based on mutation. The most effective approaches are the series of *equivalence modulo inputs* (EMI) work [66, 67, 139]. Given a seed program, EMI mutates the seed program by removing/changing dead regions or inserting code into live regions. EMI guarantees that the mutated program has the same semantics *w.r.t.* the same input. CsmithEdge[31] removes "safe math" wrappers from Csmith programs to lift the expressiveness constraints. GrayC [32] mutates seed programs with coverage guidance. This approach is effective in finding compiler crashes. However, no miscompilation could be found.

Similar to the generators, they are also limited by the designed mutation rules and the seed programs. Although our approach is based on mutation, the large amount of available real-world code provides us with more elevated diversity in generating mutants.

Using real-world code for compiler testing. There are indirect usages of real-world code by using machine learning models for compiler testing. DeepSmith [26] trains an OpenCL program generator on a large set of real-world code. Fuzz4All [156] utilizes large-language models to produce programs. One fundamental limitation of these approaches is that they do not guarantee the validity of generated programs. Instead of using models, LangFuzz [55] aims at fuzzing JavaScript interpreters and uses code fragments from bug reports. FreeFuzz [153] collects API information from deep learning libraries and then synthesizes API calls to test deep learning compilers. Our work differentiates from these efforts by semantically fusing real-world code with the seed program and ensuring program validity.

Program synthesis. Much work has been done in the domain of program synthesis [50]. The goal is to automatically find a program that satisfies the user-intended specifications such as logical constraints [98] and input/output examples [115]. The purpose of program synthesis is not testing compilers but rather aiding human programmers, education, *etc.*

7 CONCLUSION AND FUTURE DIRECTIONS

In this thesis, we have introduced novel approaches for detecting various software defects at different levels in the software development pipeline: CompDiff, PGE, and Sand at the *code* level, UBfuzz at the *code analysis* level, and Creal at the *code compilation* level. These tools have identified many real-world bugs in critical software systems.

CompDiff, presented in Chapter 2, is a simple, straightforward, yet effective approach for finding unstable code in C/C++ programs. CompDiff concerns program input/output behaviors across metamorphic compiler implementations. The succinct design of CompDiff poses no constraint on the underlying programming language and is generally applicable. We also integrated CompDiff into AFL++ to improve its practicality. Our extensive evaluation on both benchmark and real-world programs confirmed that CompDiff is effective in covering a broad range of unstable code and significantly complements existing sanitizers by finding many unique bugs. We expect our study to inspire the community further to explore the impact and the detection of unstable code.

PGE, presented in Chapter 3, is a novel technique for boosting greybox fuzzing's efficiency and bug detection. The high-level insight is that partial test execution can help separate interesting and non-interesting tests, thus a fuzzer can terminate those non-interesting executions early for higher fuzzing throughput. We have empirically shown that most test inputs during fuzzing are non-interesting, and execution prefixes can help select interesting tests. As a proof-of-concept, we have integrated PGE into AFL++. Our results show that AFL++-PGE improves not only AFL++'s performance but, more importantly, also its coverage-increasing and bug-finding capability. PGE is general and, in principle, can enhance any greybox fuzzer. This work provides a simple, effective realization and motivates future explorations of this direction.

Sand, presented in Chapter 4, is a new fuzzing framework to decouple sanitization from the fuzzing loop. Sand performs fuzzing on the normally built program and only executes sanitizer-enabled programs when input is identified as sanitization-required. Sand utilizes the fact that most of the fuzzer-generated inputs do not need sanitization, which enables it to spend most of the fuzzing time on the normally built program. To identify sanitization-required inputs, we have designed a practical and effective execution analysis via the execution pattern.

UBFUZZ, presented in Chapter 5, is the first validation framework for testing sanitizer implementations. UBFUZZ contains a UB program generator that generates UB programs from a seed program via shadow statement insertion. Based on this generator, we have employed differential testing across multiple compilers to test sanitizers. To filter out discrepancies caused by compiler optimizations, we have designed a new test oracle, crash-site mapping, that is capable of accurately identifying true sanitizer bugs. UBFUZZ has discovered 31 bugs in ASan, UBSan, and MSan from both GCC and LLVM. Our work represents a promising initial step toward comprehensive validations of sanitizer implementations and highlights the importance of this problem.

CREAL, presented in Chapter 6, is a novel approach to boost the expressiveness of existing random program generators by fusing real-world code. We first construct a function database by collecting functions from real-world applications. We then augment each function with input/output pairs to model function semantics precisely. With the function database, we synthesize function calls and inject them into seed programs. We have demonstrated the effectiveness of CREAL by identifying 132 bugs, with 121 confirmed and 97 fixed in the two most popular C compilers, GCC and LLVM. Our innovative approach of blending real-world code with synthetic programs offers a compelling and optimistic research direction for testing compiler implementations. From a technical standpoint, discovering more effective methods for extracting code could enhance the variety and versatility of generated programs. From an application standpoint, further research is needed to explore how this approach can be applied to other language compilers.

Future work. Next, we discuss several interesting and exciting future work on extending our work.

7.1 COMPDIFF BEYOND C/C++

COMPDIFF was initially designed for detecting undefined behaviors in C/C++ programs. However, its design stems from the common consequence of undefined behaviors, i.e., the language specification poses no restrictions on their runtime behaviors. Since compilers implemented the language specification, they may do arbitrary or divergent translations or optimizations on code with undefined behaviors. In general, undefined behaviors in other languages, such as Rust, also share this common consequence. It is thus interesting to try whether or not the principled detection methodology of CompDiff can be applied for detecting undefined behavior in other program languages.

7.2 FAULTY EXECUTION DETECTION BASED ON PGE AND SAND

Given an input or its execution, deciding whether or not this input or the execution triggers a bug in the program is challenging. This is the so-called *test oracle problem*. Solving the test oracle problem or detecting faulty executions for general programs is hard while demanding. The general philosophy behind PGE and Sand provides a possible way toward it. The common philosophy behind them is that program executions contain rich information that can be utilized to infer many properties regarding the executions themselves. From these two works, we believe that *analyzing the dynamic execution of an input can help us decide whether or not this input is bug-triggering.* We used the code access pattern in both PGE and Sand to analyze executions. Despite their simplicity, we have successfully inferred certain properties of an execution. Therefore, we believe that by utilizing advanced analysis, such as extracting more information or applying machine learning techniques, we can precisely detect faulty executions and thus move toward the test oracle problem.

7.3 VALIDATING BUG DETECTORS VIA UBFUZZ

To validate sanitizer implementations, our UBfuzz has two core components, *i.e.*, a program generator and a test oracle. While the test oracle is tailored for sanitizers, the program generator, designed for automatically producing complex programs with various types of undefined behaviors, is general. Countless efforts have been put in both academia and industry to detect undefined behaviors in C/C++ programs. Their reliability is essential for the overall reliability of many software systems that rely on them. UBfuzz thus can be extended to validate them extensively. For dynamic tools such as Valgrind [107], similar test oracles to UBfuzz may be required. For static tools such as Infer [104], UBfuzz's program generator is directly applicable or can be applied with minor changes.

7.4 UNIVERSAL PROGRAM GENERATION BASED ON CREAL

Due to the substantial importance of compilers, significant research efforts have been put into compiler testing. Despite the existence of many programming languages, a large proportion of compiler testing efforts were spent on popular languages such as C/C++ and Java. Researchers devote countless efforts to designing random program generators to produce diverse, expressive, and valid programs for stress-testing compilers. Implementing a program generator requires a lot of research and engineering resources. Still, all current rule-based program generators, such as Csmith [159], are constrained by hard-coded rules as discussed in Chapter 6. Designing better program generators becomes harder. Even worse, many programming languages do not have program generators to test their implementations.

Creal provides an alternative path toward program generation. The initial positioning of Creal is a new way to augment existing random program generators by injecting real-world code. However, the core message of Creal can also be interpreted as *we can generate a new program by blending many code snippets.* For example, given any seed program or even a function from Creal's function database, we can iteratively inject other code snippets or functions into it. This technical routine is not only suitable for C/C++ but potentially also applicable to a broader range of programming languages, such as Rust and Javascript.

BIBLIOGRAPHY

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Announcing OSS-Fuzz: Continuous fuzzing for open source software. *Google Testing Blog*, 2016.

[2] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. Exploiting the saturation effect in automatic random testing of android applications. In *Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft'15, pages 33–43, 2015.

[3] Austin Appleby. Murmurhash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3, 2016. Accessed: March 21, 2024.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS'19, pages 1–15, 2019.

[5] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS'22, pages 351–364, 2022.

[6] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'20, pages 713–724, 2020.

[7] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'20, pages 678–689, 2020.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 2329–2344, 2017.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 1032–1043, 2016.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[11] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'17, pages 117–128, 2017.

[12] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. FUZZOLIC: Mixing fuzzing and concolic execution. *Computers and Security*, 108, 2021.

[13] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'11, pages 213–223, 2011.

[14] C-Standards. Iso/iec 9899:2018, programming languages — C. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf, 2018. Accessed: March 21, 2024.

[15] C++-Standards. Standard for programming language C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf, 2020. Accessed: March 21, 2024.

[16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 2095–2108, 2018.

[17] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys*, 53(1), 2020.

[18] Liming Chen and Algirdas Avizienis. N-version Programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing*, FTCS'78, pages 3–9, 1978.

[19] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, SP'18, pages 711–725, 2018.

[20] 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, pages 499–513, 2019.

[21] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. MEUZZ: Smart seed scheduling for hybrid fuzzing. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'20, pages 77–92, 2020.

[22] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP'20, pages 1580–1596, 2020.

[23] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'16, pages 85–99, 2016.

[24] CISQ. The cost of poor software quality in the US: A 2022 report. https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf, 2022. Accessed: March 21, 2024.

[25] Developers of CppCheck. A tool for static C/C++ code analysis. http://cppcheck.sourceforge.net/, 2022. Accessed: March 21, 2024.

[26] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'18, pages 95–105, 2018.

[27] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. AnghaBench: A suite with one million compilable c benchmarks for code-size reduction. In *Proceedings of the 19th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'21, pages 378–390, 2021.

[28] Mila Dalla Preda, Roberto Giacobazzi, Arun Lakhotia, and Isabella Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'15, pages 329–341, 2015.

[29] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP'20, pages 1497–1511, 2020.

[30] Zhen Yu Ding and Claire Le Goues. An empirical study of OSS-Fuzz bugs. In *Proceedings of the 2021 IEEE/ACM International Conference on Mining Software Repositories*, MSR'21, pages 131–142, 2021.

[31] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE'20, pages 1219–1223, 2021.

[32] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. GrayC: Greybox fuzzing of compilers and analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'23, pages 1219–1231, 2023.

[33] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21, pages 2829–2846, 2021.

[34] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, 2020.

[35] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. American Fuzzy Lop plus plus (AFL++-v4.01c). https://github.com/AFLplusplus/AFLplusplus/releases/tag/4.01c, 2022. Accessed: March 21, 2024.

[36] Project QMK Firmware. Quantum mechanical keyboard firmware. https://github.com/qmk/qmk_firmware/blob/gb_port/keyboards/bfake/matrix.c, 2023. Accessed: March 21, 2024.

[37] Project FreeBSD. FreeBSD. https://github.com/freebsd/freebsd-src/blob/stable/10/contrib/binutils/libiberty/memset.c, 2023. Accessed: March 21, 2024.

[38] Project FreeType. FreeType. https://github.com/freetype/freetype/blob/master/src/tools/ftrandom/ftrandom.c, 2023. Accessed: March 21, 2024.

[39] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20, pages 2577–2594, 2020.

[40] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, SP'18, 2018.

[41] Developers of GCC. Passes and files of the compiler. https://gcc.gnu.org/onlinedocs/gccint/Passes.html, 2023. Accessed: March 21, 2024.

[42] Developers of GCC. Options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2024. Accessed: March 21, 2024.

[43] GCC-Bug. Bug report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105714, 2022.

[44] GCC-Bug. Bug report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=106558, 2022.

[45] GCC-Bug. Bug report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108085, 2022.

[46] GCC-Bug. Bug report. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109151, 2023.

[47] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential regression testing for rest apis. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'20, pages 312–323, 2020.

[48] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE'17, pages 50–59, 2017.

[49] Google. OSS-Fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2022. Accessed: March 21, 2024.

[50] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[51] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Conference on Security*, USENIX Security'13, pages 49–64, 2013.

[52] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3), 2020.

[53] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'21, pages 230–243, 2021.

[54] Adrian Herrera, Mathias Payer, and Antony L. Hosking. DatAFLow: Toward a data-flow-guided fuzzer. *ACM Transactions on Software Engineering and Methodology*, 32(5), 2023.

[55] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, USENIX Security'12, 2012.

[56] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient N-version execution framework. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 339–353, 2015.

[57] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security, Workshop on Binary Analysis Research*, 2018.

[58] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP'20, pages 1613–1627, 2020.

[59] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DIFUZZRTL: Differential fuzz testing to find cpu bugs. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy*, SP'21, pages 1286–1303, 2021.

[60] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? An empirical study of coverity usage. In *Proceedings of the IEEE 30th International Symposium on Software Reliability Engineering*, ISSRE'19, pages 323–333, 2019.

[61] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. Don't Look UB: Exposing sanitizer-eliding compiler optimizations. *Proceedings of the ACM Programming Languages*, 7(PLDI), 2023.

[62] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, pages 249–263, 2020.

[63] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. Igor: Crash deduplication through root-cause clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS'21, pages 3318–3336, 2021.

[64] Edward L Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 53(282):457–481, 1958.

[65] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 2123–2138, 2018.

[66] 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, pages 216–226, 2014.

[67] 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'15, pages 386–399, 2015.

[68] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE'18, pages 475–485, 2018.

[69] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[70] Xavier Leroy. CompCert. https://compcert.org/index.html, 2024. Accessed: March 21, 2024.

[71] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. Partisan: fast and flexible sanitization via run-time partitioning. In *Proceedings of the 21st International SymposiumResearch in Attacks, Intrusions, and Defenses*, RAID'18, pages 403–422, 2018.

[72] Shaohua Li. Artifact for CompDiff. https://doi.org/10.5281/zenodo.7612226.

[73] Shaohua Li. Artifact for Creal. https://doi.org/10.5281/zenodo.10802596.

[74] Shaohua Li. Artifact for PGE. https://doi.org/10.5281/zenodo.7727577.

[75] Shaohua Li. Artifact for UBfuzz. https://doi.org/10.5281/zenodo.8406414.

[76] Shaohua Li and Zhendong Su. Accelerating fuzzing through prefix-guided execution. *Proceedings of the ACM Programming Languages*, 7(OOPSLA), 2023.

[77] Shaohua Li and Zhendong Su. Finding unstable code via compiler-driven differential testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'23, pages 238–251, 2023.

[78] Shaohua Li and Zhendong Su. UBFuzz: Finding bugs in sanitizer implementations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'24, 2024.

[79] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'17, pages 627–637, 2017.

[80] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Conference on Security Symposium*, USENIX Security'21, 2021.

[81] Developers of LibTooling. LibTooling. `https://clang.llvm.org/docs/LibTooling.html`, 2023. Accessed: March 21, 2024.

[82] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'15, pages 65–76, 2015.

[83] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'22, 2022.

[84] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS'21, pages 487–502, 2021.

[85] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM Programming Languages*, 4(OOPSLA), 2020.

[86] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proceedings of the ACM Programming Languages*, 7(PLDI), 2023.

[87] Developers of LLDB. The LLDB Debugger. https://lldb.llvm.org/, 2023. Accessed: March 21, 2024.

[88] Developers of LLVM. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022. Accessed: March 21, 2024.

[89] Developers of LLVM. LLVM's analysis and transform passes. https://llvm.org/docs/Passes.html, 2023. Accessed: March 21, 2024.

[90] Developers of LLVM. Clang – The Clang C, C++, and Objective-C compiler. https://clang.llvm.org/docs/CommandGuide/clang.html, 2024. Accessed: March 21, 2024.

[91] LLVM-Bug. Bug report. https://github.com/llvm/llvm-project/issues/55189, 2022.

[92] LLVM-Bug. Bug report. https://github.com/llvm/llvm-project/issues/60236, 2023.

[93] LLVM-Bug. Bug report. https://github.com/llvm/llvm-project/issues/61982, 2023.

[94] LLVM, Developers of. SanitizerCoverage. https://releases.llvm.org/11.0.1/tools/clang/docs/SanitizerCoverage.html, 2021. Accessed: March 21, 2024.

[95] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium*, USENIX Security'19, pages 1949–1966, 2019.

[96] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[97] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE'20, pages 1024–1036, 2020.

[98] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[99] Nathan Mantel et al. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemother Rep*, 50(3):163–170, 1966.

[100] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with data dependency information. In *Proceedings of the 2022 IEEE European Symposium on Security and Privacy*, EuroSP'22, 2022.

[101] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? *Proceedings of the ACM Programming Languages*, 3(OOPSLA), 2019.

[102] Björn Mathis, Rahul Gopinath, and Andreas Zeller. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'20, pages 27–37, 2020.

[103] Patrick E McKnight and Julius Najab. Mann-Whitney U test. *The Corsini Encyclopedia of Psychology*, 2010.

[104] Meta. A tool to detect bugs in Java and C/C++/Objective-C code. https://fbinfer.com/, 2022. Accessed: March 21, 2024.

[105] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, SP'19, pages 787–802, 2019.

[106] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS'21, pages 351–365, 2021.

[107] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100, 2007.

[108] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. DifFuzz: differential fuzzing for side-channel analysis. In *Proceedings of the ACM/IEEE 41st International Conference on Software Engineering*, ICSE'19, pages 176–187, 2019.

[109] NIST. Juliet test suite for C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112, 2017. Accessed: March 21, 2024.

[110] NIST. CWE-469: Use of pointer subtraction to determine size. https://cwe.mitre.org/data/definitions/469.html, 2023. Accessed: March 21, 2024.

[111] Yannic Noller, Rody Kersten, and Corina S Păsăreanu. Badger: Complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'18, pages 322–332, 2018.

[112] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. HyDiff: Hybrid differential software analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE'20, pages 1273–1285, 2020.

[113] Community of NIST. CWE: Weaknesses in software written in C. https://cwe.mitre.org/data/definitions/658.html. Accessed: March 21, 2024.

[114] Community of NIST. CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023. Accessed: March 21, 2024.

[115] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'15, pages 619–630, 2015.

[116] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*, USENIX Security'20, pages 2289–2306, 2020.

[117] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering*, ICSE'17, pages 609–620, 2017.

[118] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, SP'18, pages 697–710, 2018.

[119] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 2017 IEEE Symposium on security and privacy*, SP'17, pages 615–632, 2017.

[120] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A greybox fuzzer for network protocols. In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification*, ICST'20, pages 460–465, 2020.

[121] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. Music: Mutation analysis tool with high configurability and extensibility. In *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, ICSTW'18, pages 40–46, 2018.

[122] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[123] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium*, NDSS'17, pages 1–14, 2017.

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

[125] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'21, pages 142–157, 2021.

[126] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'20, pages 1140–1152, 2020.

[127] Coverity Scan. Coverity scan: Find and fix defects in your Java, C/C++, C#, JavaScript, Ruby, or Python open source project for free. `https://scan.coverity.com/`, 2022. Accessed: March 21, 2024.

[128] David A Schum. *The evidential foundations of probabilistic reasoning*. Northwestern University Press, 2001.

[129] Mozilla Security. Fuzzdata. `https://github.com/MozillaSecurity/fuzzdata`, 2021. Accessed: March 21, 2024.

[130] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 309–318, 2012.

[131] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Proceedings of the 2016 IEEE Cybersecurity Development*, SecDev'16, pages 157–157, 2016.

[132] Kostya Serebryany. Sanitize, fuzz, and harden your C++ code. In *Proceedings of the 2016 USENIX Enigma*, 2016.

[133] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 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'20, pages 737–749, 2020.

[134] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, SP'19, pages 803–817, 2019.

[135] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20, pages 2541–2557, 2020.

[136] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proceedings of the 13th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'15, pages 46–55, 2015.

[137] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, NDSS'16, pages 1–16, 2016.

[138] Project GB Studio. GBStudio. https://github.com/chrismaltby/gb-studio/blob/develop/buildTools/linux-x64/mod2gbt/mod2gbt.c, 2023. Accessed: March 21, 2024.

[139] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'16, pages 849–863, 2016.

[140] Developers of Tcpdump. Tcpdump. https://github.com/the-tcpdump-group/tcpdump, 2022. Accessed: March 21, 2024.

[141] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'22, pages 697–709, 2022.

[142] Kaushik Veeraraghavan, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 369–384, 2011.

[143] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, SP'15, pages 866–879, 2015.

[144] Jonas Benedict Wagner. *Elastic program transformations: Automatically optimizing the reliability/performance trade-off in systems software*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2017.

[145] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE'20, pages 999–1010, 2020.

[146] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'19, pages 1–15, 2019.

[147] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, SP'17, pages 579–594, 2017.

[148] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE'18, pages 61–64, 2018.

[149] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. Odin: On-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'22, pages 1010–1024, 2022.

[150] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. Compilation consistency modulo debug information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'23, pages 146–158, 2023.

[151] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSys'12, pages 1–7, 2012.

[152] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 260–275, 2013.

[153] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE'22, pages 995–1007, 2022.

[154] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[155] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering*, ICSE'20, pages 789–800, 2020.

[156] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. In *Proceedings of the 46th International Conference on Software Engineering*, ICSE'24, pages 1–13, 2024.

[157] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 2313–2328, 2017.

[158] Developers of xxHash. xxhash: Extremely fast hash algorithm. `https://github.com/Cyan4973/xxHash`, 2016. Accessed: March 21, 2024.

[159] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, pages 283–294, 2011.

[160] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, USENIX Security'18, pages 745–761, 2018.

[161] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2014. Accessed: March 21, 2024.

[162] Michał Zalewski. Fuzzing random programs without execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014. Accessed: March 21, 2024.

[163] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'21, 2021.

[164] Qian Zhang, Jiyuan Wang, and Miryung Kim. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'21, pages 242–254, 2021.

[165] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security'22)*, pages 4345–4363, 2022.

[166] Su Zhendong. Emi compiler testing. https://people.inf.ethz.ch/suz/emi/index.html, 2023. Accessed: March 21, 2024.

[167] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE'20, pages 858–870, 2020.

[168] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. An empirical study of optimization bugs in gcc and llvm. *Journal of Systems and Software*, 174, 2021.

[169] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys*, 54(11s):1–36, 2022.

[170] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20, pages 2255–2269, 2020.

CURRICULUM VITAE

Personal Data

Name	Shaohua Li
Date of Birth	8 December, 1994
Citizen of	China

Education

2019-2024	Ph.D. in Computer Science ETH Zurich Zürich, Switzerland
2016-2019	Master of Engineering University of Science and Technology of China Hefei, China
2012-2016	Bachelor of Engineering University of Science and Technology of China Hefei, China