BETTER GENERATIVE COMPILER FUZZING FOR UNSAFE LANGUAGES

by

Vsevolod Livinskii

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Kahlert School of Computing

The University of Utah

December 2024

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Vsevolod Livinskii**

has been approved by the following supervisory committee members:

John Regehr	, Chair	**11/7/2024**
		Date Approved
Eric Norman Eide	, Member	**11/22/2024**
		Date Approved
Konstantin Serebryany	, Member	**11/11/2024**
		Date Approved
Pavel Panchekha	, Member	**11/7/2024**
		Date Approved
Zvonimir Rakamaric	, Member	**12/04/2024**
		Date Approved
Dmitry Babokin	, Non-voting member	

and by **Mary W. Hall**, Director of

the **Kahlert School of Computing**

and by **Darryl P. Butt**, Dean of the Graduate School.

ABSTRACT

Compilers are part of the foundation upon which software systems are built, so they must be as correct as possible. Compiler fuzzing is a technique that has achieved impressive results in finding compiler bugs. This dissertation presents a novel approach to constructing generative compiler fuzzers for unsafe languages. Techniques developed in this dissertation helped me to find hundreds of bugs in GCC, LLVM, the Intel C/C++ Compiler, and other tools. My first main contribution is a novel static undefined behavior avoidance mechanism. I use it to generate tests that are compliant with the language standard, allowing me to detect miscompilation errors. My second main contribution is a novel mechanism to target compiler optimizations explicitly. It allows me to thoroughly test various scalar and loop optimizations, including those found in compilers for data-parallel languages. This technique increases the diversity of generated tests and helps me detect difficult-to-find bugs. These ideas were implemented in the YARPGen compiler fuzzer, which is used by multiple companies and research groups.

To friends and family,

who have supported me on this long journey.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the unwavering support, guidance, and encouragement of several remarkable individuals.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. John Regehr, for his exceptional mentorship throughout this journey. His thoughtful feedback, deep insights, and invaluable assistance tremendously supported me during each step of this endeavor.

I am incredibly grateful to Dmitry Babokin for sharing his knowledge, providing valuable feedback throughout my Ph.D., and being my mentor during internships at Intel®. His guidance has been crucial to my growth both academically and professionally.

I would also like to thank Prof. Eric Eide, Konstantin Serebryany, Prof. Pavel Panchekha, Prof. Zvonimir Rakamaric, and Dmitry Babokin for serving as committee members. Their insightful feedback and constructive guidance played a significant role in shaping the direction and quality of this dissertation.

I sincerely appreciate the cooperation and support of the GCC, LLVM, Intel® Implicit SPMD Program Compiler, and Intel® oneAPI DPC++ compiler development teams. Without them, achieving such great practical results would not be possible. In particular, I am especially thankful to Martin Liška, Richard Biener, and Jakub Jelinek for their overwhelming assistance with GCC bugs.

I would like to thank the Flux research group at the University of Utah for providing me access to the Emulab cluster for running experiments.

Finally, I am grateful to Intel® Corporation for partially sponsoring this research, allowing me to pursue and complete this work.

CHAPTER 1

INTRODUCTION

Human society relies on various software systems to perform critical tasks. We use such systems to control all aspects of our lives, from the infrastructure of our cities to the devices that we use on a daily basis, such as our computers, phones, cars, and planes. Errors in these systems can lead to loss of life, property, and money. Therefore, software developers should put much effort into ensuring their software is correct and reliable.

Unfortunately, it is not enough to test only the application itself due to complex dependencies between it and the rest of the software stack. One of the foundational components of the software stack is the compiler. A compiler is a program that translates source code written in a programming language into a target language, such as machine code. Compiler errors can lead to bugs in generated code, causing errors in the operating system, dependencies, libraries, or the application itself. Therefore, it is crucial to test compilers thoroughly.

Compiler testing is a challenging task due to the complexity of the compiler itself. For example, LLVM and GCC—two of the most popular open-source compilers—have millions of lines of code. Moreover, compilers are constantly evolving. New optimizations, architecture support, and features, such as sanitizers,[1] are added to them regularly. All such changes can introduce new bugs into the compiler and require thorough testing.

Conventional testing methods, such as test suites and uniform tests, still miss bugs. For instance, we found a bug in GCC, shown in Listing 1.1, that was not caught by the existing testing methods for almost four years. One of the alternative testing methods is random testing, also known as fuzzing.

---

[1] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

```c
int foo() {
  return -1;
}

int main() {
  int c = foo() >= 0U && 1;
  if (c != 1)
    abort ();
}
```

Listing 1.1: GCC bug #105189 that we found. It remained undetected for almost four years and affects GCC versions 9, 10, 11, and 12. The correct version initializes variable `c` to 1, while the incorrectly optimized one initializes it to 0, causing the program to abort.

This dissertation presents a new approach to compiler fuzzer construction. The rest of this chapter briefly describes the challenges of compiler testing. It then gives an overview of random testing, followed by a description of the compiler random testing technique, existing approaches, the layout of this dissertation, and a summary of my contributions.

1.1 A Compiler

A compiler is a program that translates source code written in a programming language into a target language, such as machine code. It allows software developers to use high-level languages like C++ and Java to write programs for various hardware architectures. This way, programmers can use high-level abstractions without worrying about the low-level implementation details of specific architectures. Moreover, it allows developers to use the same codebase to target multiple hardware architectures. For example, we can use the same C++ codebase to target x86 and ARM. Additionally, compilers perform various internal optimizations, such as auto-vectorization, dead code elimination, and constant propagation. These optimizations significantly improve the performance of the generated code . Finally, compilers also implement various auxiliary features, such as sanitizers and code coverage instrumentation. These features are used to detect bugs in the generated code and improve the quality of the generated code.

These features and optimizations make compilers very complex pieces of software. For instance, two of the most popular open-source compilers—GCC and LLVM—consist of

over 10 million lines of code each. Moreover, additional complexity comes from the non-trivial interaction between different components of the compiler. In particular, sanitizers can conflict with optimizations, hiding existing bugs, or introduce new ones [1]. In addition, industry-grade open-source compilers are developed by thousands of developers from all over the world. This concurrent work unavoidably leads to miscommunications that can compromise the correctness of the compiler. For example, incorrect assumptions about undefined behavior in LLVM intermediate representation (IR) [2] were caused by such errors. Also, compilers are constantly evolving to keep up with the development of hardware architectures and programming languages. For instance, compilers have to be updated each time the ISO C++ committee releases a new standard.

All this complexity and constant modification of compilers make them prone to errors. Conventional testing methods, such as test suites and uniform tests, cannot catch all the bugs in the compiler. Both GCC and LLVM have thousands of open bugs reported by users, indicating that these bugs were not caught by the existing testing methods. This number might be even higher because not all users report bugs that they find. Often, they want to avoid the hassle of reporting a bug and are satisfied with a workaround that works for them. One way to detect missed errors and improve the compiler's robustness is to use random testing, also known as fuzzing.

1.2 Random Testing

Fuzzing is an automated software testing technique. It automatically generates random inputs to the software under test (SUT) and checks whether the SUT crashes or produces incorrect results. In contrast with conventional testing methods, fuzzing can explore the test space without any human input.

Conventional testing methods, such as test suites and uniform tests, are based on a fixed set of inputs. They will always explore the same execution path within the SUT. Therefore, they will always find the same bugs (or nothing). Bugs that can be found by conventional testing methods will be found after the first execution of the tests. Subsequent executions will not give us any new information about the SUT.

In the case of random testing, each execution of a new test produced by a fuzzer is likely to explore a new execution path within the SUT. Therefore, we can increase the number of explored execution paths by increasing the number of tests. This property allows us to automate the testing process and scale the bug-finding ability of a fuzzer with the increase in computing power. Random testing has been used for decades to test various software components. It has been applied to compiler testing for more than 60 years [3].

1.3 Compiler Random Testing

Compiler random testing, also known as compiler fuzzing, can detect all sorts of bugs in the compiler. There are several ways compiler errors can manifest themselves. They can be divided into the following categories:

- *Internal Compiler Error (ICE)* or *crash.* The compiler crashes during the compilation of the test. Such errors can be caused by the violation of an assertion inside the compiler. This happens when the provided test violates an assumption left by a developer as a precaution. This kind of error can also happen when a compiler encounters an invalid execution state during the test compilation, such as a null pointer dereference.
- *Resource exhaustion* or *freeze*. The compiler hangs or exhausts some resources, such as memory, during the test compilation. This can happen due to an infinite loop in the compiler, conflicting optimizations, or incorrectly set thresholds.
- *Miscompilation* or *wrong code bug*. The compiler generates incorrect code for the test. This can be caused by incorrect implementation of an optimization or a bug in the code generation logic. Weak optimization preconditions or typos in the compiler often cause these errors.

The ICE is the easiest to detect. This type of error produces a visible abnormal behavior during test compilation. Such bugs can be annoying and negatively affect user experience, but they are not dangerous. Miscompilation bugs, on the other hand, can compromise the functional correctness of the generated code. They silently lead to incorrect execution results of the compiled program. Such errors are often indistinguishable from the errors in the application itself, making them hard to detect. Therefore, miscompilation bugs are often considered primary targets for compiler fuzzers.

A compiler fuzzer that is designed to detect miscompilation errors in optimizations has to produce tests that satisfy two requirements: they have to be correct and expressive. The test correctness requirement is essential for unsafe languages like C and C++. These languages are infamous for their *undefined behavior* (UB). It is a type of behavior that is not defined by the language standard. Common examples of UB in C++ include a null pointer dereference, signed integer overflow, and division by zero. If the test contains UB, a correct compiler can produce any result for it, as shown in Listing 1.2. Therefore, tests that contain UB cannot be used to detect miscompilation bugs.

Tests that are not compliant with the language standard will often be rejected by the compiler's front end during the parsing step. These tests will not be able to detect errors in the other parts of the compiler, such as the middle end and back end. The middle end is responsible for architecture-independent optimizations, such as loop invariant code motion (LICM) and dead code elimination (DCE). The back end is responsible for architecture-specific optimizations and code generation. Therefore, tests rejected by the front end terminate the compilation process before reaching the middle and back-end components. Such tests cannot be used to detect errors in optimizations and code generation logic, severely limiting the bug-finding ability of the fuzzer.

Expressiveness is the second requirement for fuzzers that target bugs in compiler optimizations. It does not have a precise definition. One approximation is to say that

```
1  # include <stdio.h>
2  int main () {
3    int x = 1;
4    x = x++ + ++x;
5    printf ("%d\n", x);
6    return 0;
7  }
```

Listing 1.2: Example of a C++ program that contains undefined behavior. Line 4 contains several reads from and writes to the same variable. Their order is not defined by the C++ standard. The program, compiled with Intel® C++ Compiler, returns 5, while the program, compiler with Clang, returns 4.[2]

[2] https://godbolt.org/z/oa6vsqc1Y

expressive tests are tests that contain as many language features as possible. Another approximation is to say that expressive tests are tests that contain as many idiomatic code patterns, such as stencils, as possible. None of these definitions are complete, but they capture the essence of the expressiveness requirement. Expressive tests are more likely to trigger optimizations and code generation features under different circumstances. This ability is the main advantage of the compiler fuzzer over conventional testing methods that are based on fixed pre-defined test suites. We cannot test optimizations that we cannot trigger. Therefore, the test has to satisfy the preconditions of the optimization for it to be applied. Such preconditions often depend on the presence of specific language features and idiomatic code. The language feature requirement means that the test has to contain specific language elements that are used by the optimization. Tests that do not contain these features will be unable to trigger optimizations that rely on them. For example, we can trigger constant propagation optimization only if the test contains some constants. The idiomatic code requirement means that the test has to contain specific code patterns commonly found in real-world applications. Compilers are designed to optimize and recognize them. For instance, for the loop invariant code motion (LICM) optimization to fire, the test has to contain a loop with an invariant expression. Tests that do not contain such expressions will not be able to trigger the LICM optimization. Thus, to increase the bug-finding ability of the fuzzer, generated tests should be as expressive as possible.

The code correctness and expressiveness requirements are in tension with each other. It is easy to generate expressive tests that contain many language features, but they are likely to contain UB. The difficulty of eliminating UB increases with the number of language features in the test because each feature has its own rules for UB. For example, the addition of signed integer types to the test can cause a signed integer overflow, so the fuzzer has to avoid this kind of UB. An addition of pointers forces the fuzzer to avoid null pointer dereference in the generated tests. Thus, the number of potential UB cases increases with the number of language features in the test. In contrast, it is easy to generate tests without UB by using a safe subset of language features. However, such tests will likely not be expressive enough to trigger various optimizations and code generation features. They will miss the required language features and idiomatic code patterns. Therefore, the goal of a

fuzzer developer is to find the right balance between test correctness and expressiveness. All existing compiler fuzzer approaches try to find their own best way to do it.

1.3.1 Existing Approaches

All existing compiler random testing approaches can be divided into three categories [4]. The first one is a generation-based approach. The core idea behind it is to create new tests out of nothing. For every new seed, the fuzzer creates a new test from scratch. The second one is a mutation-based approach. Its main idea is to modify existing tests. This approach usually uses a test suite or output of the generative fuzzer as an initial corpus. The third one is a machine-learning-based approach. It is a combination of generation-based and mutation-based approaches. It uses a collection of existing tests to train a machine-learning model but still generates new tests from scratch.

For the purpose of this dissertation, we will focus on the generation-based approach. There are several reasons for that.

- This approach does not require any external initial corpus. This advantage is significant for testing compilers for emerging languages where such a corpus might not exist or be limited.
- The mutation-based approach usually relies on third-party tools, such as sanitizers, to check whether a mutation produces code that is compliant with the language standard. Generative fuzzers do not have this limitation because they are usually designed to produce tests that are always compliant with the language standard to begin with. Therefore, generative fuzzers can be used to test sanitizers and other tools rather than rely on their correctness.
- The mutation-based approach usually does not provide a precise control over language features in the produced tests. It is especially true if the fuzzer uses existing test suites as an initial corpus. The ability to control language features is crucial if we want to test compilers for emerging languages. Such compilers are often in the early stages of development, so they might not support some of the language features yet or contain known bugs. Therefore, we want to be able to produce tests without such features to keep the fuzzing loop running while avoiding known errors.

Additionally, we will also narrow our focus to the C-family languages. There are several reasons for that. First, C-family languages remain in the top ten most popular programming languages. They are widely used to create various software components, from operating systems to embedded systems. Second, such languages are often used as a base for new languages. For example, OpenCL, SYCL, DPC++, and CUDA are all based on C++ . Therefore, if we create a language-agnostic compiler fuzzer for C-family languages, we can simultaneously test compilers for all these languages.

A detailed overview of the existing solutions, both generative and mutation-based, can be found in Chapter 6. Here, we will give a brief overview of the existing generation-based approaches for C-family languages compilers and their limitations to provide a context for our work.

There exist three main techniques used by the generative fuzzers for C-family languages compilers to create tests that are compliant with the language standard: language subset, wrapper functions, and static analysis.

The first method is based on the idea of using a language subset that does not cause UB. The Quest compiler fuzzer uses this approach [5]. This fuzzer was designed to test C calling conventions, so it does not emit tests that contain any arithmetic operations, control flow, or loops. This approach works well for fuzzers that want to test specific parts of the language standard, but it cannot be used to create a general-purpose compiler fuzzer. Tests generated by this approach are not expressive enough to trigger general-purpose optimizations designed to work with real-world code.

The second approach uses wrapper functions. It was pioneered by Csmith [6] and later adopted by other compiler fuzzers, such as CUDAsmith [7] and CLSmith [8]. Csmith uses wrapper functions to avoid out-of-bounds array access and UB in arithmetic operations, such as division by zero and signed integer overflow. These functions replace the regular arithmetic operations with a runtime check for UB before performing the actual operation. An example of such a wrapper function that Csmith uses to avoid UB in a division operation is shown in the Listing 1.3. Tests generated with the wrapper functions approach will not contain plain arithmetic operations. Real-world programs are never written in such a manner. Compilers are designed to optimize user code to begin with. Therefore, if the test

```
int32_t safe_div(int32_t si1, int32_t si2) {
  if ((si2 == 0) ||
      ((si1 == INT_MIN) && (si2 == (-1)))) {
    return si1;
  }
  else {
    return (si1 / si2);
  }
}
```

Listing 1.3: An example of wrapper function that Csmith uses to avoid undefined behavior in division operation [9].

significantly differs from the real-world code, it is less likely to trigger optimizations and, thus, less likely to find bugs in them. Therefore, the wrapper function approach limits the bug-finding ability of the fuzzer, and recent research supports this claim [10, 11].

Finally, the third approach is based on the static analysis that happens during the generation process. Csmith uses whole program analysis to avoid read-after-write and write-after-write conflicts. It also allows Csmith to avoid unspecified evaluation order, e.g., in function arguments. The Orange [12–14] family of compiler fuzzers uses static analysis to avoid UB in arithmetic operations. The static analyzer developed by Orange authors allows them to produce straight-line code (i.e., code without any control flow or loops) that is compliant with the C language standard. Unfortunately, they implemented it for a restricted subset of the C language. For example, it only supports variables of basic integer types. These limitations severely limit the bug-finding ability of the Orange family fuzzers.

One of the goals of this dissertation is to develop a novel static UB elimination technique that lifts the limitations of existing approaches. We want to apply static analysis to avoid UB in a broader subset of the C-family languages. This way, we can ensure generated test compliance with the language standard without any runtime checks.

As we mentioned earlier, the ability to generate expressive tests is important for the bug-finding ability of the compiler fuzzer. The primary method to achieve this goal is to tune the probability of random choices inside the fuzzer to ensure that generated tests resemble the code of real applications as much as possible. For example, Csmith uses a fine-tuned set of probabilities to achieve it. This approach works well for the optimizations

that rely on the presence of language features that exist as first-class elements. In particular, we can increase the probability of generating a constant in arithmetic expression to increase the probability of triggering the constant propagation optimization. However, it only works well for the optimizations that rely on preconditions that have some explicit properties that can be expressed through the language specification. For instance, we cannot increase the probability of generating common subexpressions necessary for the common subexpression elimination optimization by increasing the probability of generating any language element. Therefore, the approach that is based on fine-tuning the random choices does not cover all possible optimizations and, thus, limits the bug-finding ability of the fuzzer. Methods that have been proposed to increase the expressiveness of the generated tests (e.g., swarm testing [15] or history-guided configuration diversification [16]) still operate in terms of random choice probabilities. Therefore, such methods suffer from the same limitations as the basic one. Moreover, the probability tuning approach is susceptible to the saturation effect [17]. It means that a compiler fuzzer will eventually reach a point where it will not be able to find any new bugs, no matter how many tests it generates. This happened to Csmith, and it is generally not able to find any new bugs in GCC [18].

Our goal is to develop a new mechanism that will allow us to generate expressive tests and target optimizations explicitly. We want to generate tests that will contain all the necessary preconditions for the optimizations. This way, we can improve the bug-finding ability of the fuzzer and avoid or delay the saturation effect as much as possible.

Another way to increase the expressiveness of the generated tests is to use coverage-guided fuzzing. This well-known testing technique is widely used by many general-purpose fuzzers, such as AFL [19]. The idea behind it is to use code coverage as a feedback signal to guide the testing process toward parts of SUT that still need to be executed. In this approach, new tests are generated by mutating existing ones. The fuzzer then runs the test, collects the code coverage information, and uses it to judge the quality of the test. Several compiler fuzzers, including FuzzIL [20] and GrayC [21], successfully applied this technique to compiler testing. Unfortunately, they suffer from several limitations. First, each fuzzer implements its own mutation and coverage-guided fuzzing mechanism. This leads to code duplication and forces fuzzer developers to spend time re-implementing the

same functionality. Second, the current coverage-guided fuzzers usually operate on the grammar or semantic level. This can lead to the generation of tests that are not compliant with the language standard. Thus, they require third-party tools, such as sanitizers, to detect miscompilation bugs.

Our goal is to develop a universal coverage-guided compiler fuzzing framework that will lift these limitations. We want to create a solution that will allow us to use the same mutations and coverage-guided fuzzing mechanisms for multiple fuzzers. We can use it with any generative compiler fuzzer and avoid a dependency on third-party tools. This project is currently in its early stages of development and serves primarily as a prototype. It has not yet reached the intended goals and requires further development.

Another flaw of existing compiler fuzzers is that they are designed to target only one language. For example, Csmith is designed to test only C compilers. It served as a base for CLSmith [8]—a fuzzer for OpenCL compilers—but these two implementations are disjointed. New features, introduced in CLsmith, are not propagated to Csmith. Therefore, we have to maintain two separate fuzzers to target two compilers for C-family languages.

One of the goals of this dissertation is to develop a language-agnostic compiler fuzzer for C-family languages. We think they are substantially similar enough to create a high-level intermediate representation to capture all of them. This way, we can simultaneously test multiple compilers for C-family languages. It is crucial for emerging languages, such as ISPC and DPC++, where engineering resources are limited and cannot be spent on developing a dedicated fuzzer.

Finally, the last practical reason to create yet another compiler fuzzer is that empirical evidence shows that new fuzzers tend to find new bugs. Even after Csmith reached its saturation point, several new compiler fuzzers found new bugs in GCC and LLVM [22–24]. Therefore, we know that new compiler fuzzers tend to find bugs that are missed by the existing implementations. The fuzzing development should be seen as a cooperative community effort rather than a competition. The more fuzzers we have, the more bugs we will be able to find, leading to the more robust compilers.

To sum up, we want to develop a new approach to generative compiler fuzzer construction. The new fuzzer should use a static UB elimination mechanism to produce tests that are

always compliant with the language standard. Moreover, it should include a mechanism that will allow us to target various compiler optimizations and code generation features explicitly. Finally, it should be able to target multiple C-family languages at once.

1.4 Overview and Contributions

The goal of this dissertation is to develop a new approach to generative compiler fuzzer construction. The key contributions of this dissertation are summarized below:

- Section 2.3.2 presents a novel static undefined behavior elimination mechanism for scalar code. Section 3.3.5 extends this mechanism to loops.
- Section 2.3.3 introduces the idea of generation policies that allow us to target various compiler optimizations and code generation features explicitly. It describes it in the setting of scalar code, and Section 3.3.2 extends it to loops.
- Section 3.3.6 describes the high-level intermediate representation that allows us to target several languages at the same time.

The rest of this document is laid out as follows. Chapter 2 introduces a new static undefined behavior elimination mechanism for scalar code. It also introduces generation policies for scalar code—a novel mechanism that allows us to target optimizations explicitly. Chapter 3 extends both of these ideas to loops. It also introduces a new high-level IR that allows us to target several languages at the same time. Chapter 4 introduces initial research on universal coverage-guided compiler fuzzing. This segment of the dissertation focuses on the development of a prototype. While it did not meet all the anticipated goals, it provides valuable insights into encountered pitfalls, offering a foundation for further exploration in this area of study. Chapter 5 summarizes the findings of our multi-year experience of compiler fuzzing. Chapter 6 gives an overview of the existing work. Finally, Chapter 7 concludes the dissertation.

CHAPTER 2

FUZZING SCALAR OPTIMIZATIONS

Adapted from V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Art. no. 196, Nov. 2020. DOI: 10.1145/3428264.

2.1 Introduction

Compilers should not crash, and they should not generate output that is not a faithful translation of their inputs. These goals are challenging to meet in the real world. For one thing, compilers often have significant code churn due to changing source languages, new target platforms to support, and additional requirements such as providing new optimizations. For another, since it is crucial for compilers to be as fast as possible, they use sophisticated, customized algorithms and data structures that are more difficult to get right than straightforward ones.

Although random test-case generation is an effective way to discover compiler bugs [6, 12–14, 22–24, 26], experience shows that any given random test-case generator, testing a particular compiler, will eventually reach a saturation point [17] where it finds very few new bugs. However, even after one test-case generator has reached this saturation point, the ability of a different generator to find bugs seems to be mostly unimpeded. For example, even after Csmith [6] had reached apparent saturation on the then-current versions of GCC and LLVM, other techniques were able to discover large numbers of residual bugs [22–24]. In other words, a test-case generator reaches a saturation point not because the compiler is bug-free, but rather because the generator contains biases that make it incapable of exercising specific parts of the compiler implementation.

This dissertation presents YARPGen, a new random test-case generator for C and C++ compilers that we created to explore several research questions. We will refer to the version of YARPGen used in this chapter as YARPGen v.1. This chapter focuses on the C and C++ scalar optimizers and investigates the following questions:

- Is it possible to generate expressive random programs that are free of undefined behaviors without resorting to the kind of heavy-handed dynamic safety checks that Csmith used?
- Is it possible to build mechanisms into a generator that help it target different parts of an optimizer?
- Do these mechanisms help YARPGen v.1 find bugs that other generators have missed? Do they help it to avoid saturation effect?

YARPGen v.1 is based on the idea that a generator should support *generation policies*: mechanisms for systematically altering the character of the emitted code, with the goal of making it possible for the generator to target optimizations explicitly. YARPGen v.1's other main contribution is to emit code that is free of undefined behavior without resorting to the heavy-handed wrapper approach employed by Csmith [6] or staying within a narrow subset of the language, as generators such as Orange [12–14] have done. So far, this generator has found 83 new bugs in GCC, 62 in LLVM, and 76 in the Intel® C++ Compiler.

The remainder of the chapter is structured as follows. Section 2.2 briefly introduces random testing for C and C++ compilers. Section 2.3 describes our approach to random differential compiler testing. Section 2.4 presents the results of our testing campaign. Section 2.5 concludes this chapter.

2.2 Background

In this dissertation, we are concerned with two kinds of compiler bugs. First, the compiler might crash or otherwise terminate without generating object code. We call this an *internal compiler error*, or ICE. Second, the compiler may generate compiled object code and terminate normally, but the generated code does not perform the right computation. We call this a *wrong code bug*, or a *miscompilation*. Compilers can go wrong in other ways,

such as looping forever, but in our experience, these failures are less common, and anyhow, the testing campaign described in this dissertation did not pursue them.

2.2.1 Random Differential Testing

Like several previous compiler testing efforts [6, 27], YARPGen v.1 is intended to be used for differential testing [28]: it generates a program that follows the rules for C and C++ programs in such a fashion that when the random program is executed, it must produce a specific output. It turns out that YARPGen v.1 has enough information about the generated program that it could generate totally self-contained test cases that check their own computation against the correct result. A disadvantage of that approach is that it makes it very hard to use an automated test case reduction using a tool such as C-Reduce [29], since many reduction steps would require a coordinated change to both the random computation and the expected answer. Differential testing avoids this problem. If independent compiler implementations often contained correlated bugs that caused them to return the same wrong answer, differential testing would not be a good idea. Empirically, this happens very rarely, and even if two compilers had the same bug, it would be noticed when a third compiler was added to the mix.

2.2.2 Undefined Behavior, Unspecified Behavior, and Implementation-Defined Behavior

A significant practical problem with using random testing to find miscompilation bugs is that C and C++ are *unsafe* programming languages: when a program executes an erroneous action such as touching heap storage that has been freed, the C or C++ implementation does not typically flag the violation by throwing an exception or terminating the program. Rather, the erroneous program may continue to execute, but with a corrupted memory state. Thus, randomly generated test cases that are erroneous are useless for differential testing. There are several hundred kinds of untrapped errors in C and C++ that are collectively referred to as *undefined behaviors* (UBs). All of these must be avoided in randomly generated code.

A related but less serious class of program behaviors is *unspecified behaviors* where the C or C++ implementation is allowed to choose one of several alternatives, but there is no requirement for consistency. An example is the evaluation order of arguments to a function. Randomly generated programs can include unspecified behaviors, but their final answer should not depend on which behavior the compiler picked.

Finally, C and C++ have many *implementation-defined behaviors* where the compiler is allowed to make a choice, such as the range of values that may be stored in a variable of `int` type, but the behavior must be documented and consistent. Since it is impractical to avoid relying on implementation-defined behaviors in C and C++ code, programs generated by YARPGen v.1 do the same thing that basically all real C and C++ applications do: they rely on a subset of these behaviors that—for a given target platform—are reliably common across all modern implementations of these languages.

2.3 Generating Expressive Scalar UB-Free Programs

This section describes how YARPGen v.1 can randomly generate loop-free C and C++ that is expressive and also statically and dynamically compliant with the rules specified in the language standards.

2.3.1 Avoiding Undefined Behavior

Programs produced by YARPGen v.1 must never execute operations that have undefined behavior. We can divide the undefined behaviors in C and C++ into those that are easy to avoid and those that are hard to avoid.

The UBs that are easy to avoid generally correspond to surface-level, static properties of generated code. For example, the requirement that a pointer is never converted to a type other than an integer or pointer type is easy to avoid because YARPGen v.1's type checker disallows code that would violate this constraint. Similarly, we avoid using uninitialized data by construction, since variables are initialized at their point of definition. Many undefined behaviors are avoided simply by not generating the program constructs that would lead them to happen. For instance, we can omit use of preprocessor in the generated code to satisfy the requirement that standard header files should not be included

when a macro with the same name as a keyword is defined. Similarly, restrictions on use of `setjmp` and `longjmp` cannot be violated because we do not generate those constructs at all.

UBs that are hard to avoid are the ones that depend on dynamic characteristics of the generated code execution, such as integer overflows. There are a number of possible strategies for achieving this goal:

- Generate tests in a fashion that is syntactically guaranteed to avoid UB, for example, by not generating arithmetic or pointers. Quest [5] used this method, which we rejected because it significantly limits the expressiveness of the generated code.
- Generate code without static constraints, inserting dynamic checks as needed to guard against undefined behaviors. Csmith [6] used this method, which we believe sacrifices expressiveness by obscuring potentially optimizable patterns behind ubiquitous safety checks. The recent research supports this claim [10, 11].
- Generate programs without worrying about undefined behaviors, and then use a checking tool to discard those that are undefined. This has worked well when generated programs are quite small [23], but YARPGen v.1 is intended to generate large programs to amortize compiler startup costs. The resulting programs would almost always be undefined.
- Run static analyses to conservatively avoid undefined behaviors during generation. Csmith used this strategy for pointer-related UBs, and the Orange family of generators [12–14] used it pervasively.

YARPGen v.1 uses the last strategy: it interleaves analysis and code generation in order to reliably and efficiently generate code that is free of UB, without unnecessarily sacrificing expressiveness. The strategy is to perform a very limited form of backtracking while generating expressions, converting operations that trigger undefined behavior into similar operations that are safe. This is described in detail below.

2.3.2 Generating Code

2.3.2.1 Creating the environment. First, YARPGen v.1 creates a collection of composite data types that will be used by the generated code, storing them into a *type table*. Types are created iteratively: the first array or struct can only contain elements with primitive types

(signed and unsigned varieties of all integer types, pointers, and—for C++—bools), but subsequent structs and arrays can contain elements of previously generated types. Some structs contain bitfields. The rules that guide bitfield generation are different for C and C++. Specifically, in C they can only have `int` or `unsigned` type and their size should be less than the size of the base type. C++ lacks these limitations. YARPGen v.1 respects each language's rules when generating code in that language.

Next, YARPGen v.1 generates a set of global variables with primitive and composite types, storing them in a *symbol table*. Each global variable belongs to one of these categories:

- inputs: these variables are read but never written after initialization; their value is constant during test case execution
- outputs: these variables are written but never read by the test function
- mixed: these are both read and written.

Input and output variables simplify UB-avoidance (Section 2.3.1). Some input variables are randomly `const`-qualified. When YARPGen v.1 is used to generate C++ code, some struct members are randomly given the `static` qualifier.

2.3.2.2 Creating statements. Each time YARPGen v.1 runs, it creates a set of—usually very large—functions. During generation, these functions are stored in a custom intermediate representation that supports static analysis and incremental construction.

The generation of each test function is top-down: a test function consists of a single top-level block, which contains a list of statements. Each statement does one of the following:

- Declares a new local variable of randomly chosen integer type, assigning it an initial value from a random arithmetic expression.
- Assigns the result of a random arithmetic expression into an existing variable.
- Begins a new conditional. Each conditional results in the generation of a random Boolean expression and one or two new blocks of statements. Once the block nesting depth reaches a configurable limit, generation of additional conditionals is disabled.

2.3.2.3 Creating expressions. YARPGen v.1's expressions are generated top-down. They are free of side effects such as embedded assignment operators (e.g., `5 + (y = 3)`), function calls, and the `++` and `--` operators. Our motivation for omitting some of these language

features is that they are lowered away early during compilation, whereas we are most interested in stress-testing middle-end compiler optimizations, which never see these language-level operators. A more detailed discussion of the limitations can be found in Section 2.3.6.

Each node in an expression is randomly one of the following:

- a unary, binary, or ternary operator,
- a type cast,
- a common subexpression (discussed in Section 2.3.3),
- a data element: a local or global variable, a struct member, an array element, a dereferenced pointer, or a constant.

Recursive generation sometimes self-limits, for example, when a leaf node is selected randomly. Otherwise, when a configurable expression depth limit is reached, the generator forces a leaf to be selected.

2.3.2.4 Avoiding UB in expressions. An expression is a tree whose leaves are variables and constants. A randomly-generated expression is likely to execute one or more operations with undefined behavior when it is evaluated. YARPGen v.1's procedure for avoiding UB works as follows.

First, the expression is type-checked in bottom-up fashion. This is done not to satisfy the C and C++ type checkers (every expression that we generate passes these checkers), but instead because the type of every subexpression needs to be known before we can analyze the flow of values through the expression. Second, again bottom-up, the values being processed by the expression are analyzed and expressions are rewritten as necessary to avoid UB.

Input variables are easy to analyze: they always hold their initial values. Output variables are also easy: they never appear in expressions. Mixed variables are updated by assignments, so their values must be tracked. Values are tracked concretely: YARPGen v.1 does not use a conservative abstract interpreter. This is possible because the generated code is loop-free (but also see Section 2.3.5).

When YARPGen v.1 detects an operation with undefined behavior, it performs a local fix, rewriting the operation into a form that does not trigger UB. The rewrites are shown in Table 2.1. For example, assume that the expression `-x` has the type `long` and `x` is a variable holding the value `LONG_MIN`. Since this expression is unsafe and must not be executed by a test program, YARPGen v.1 rewrites it as `+x`. We ensured that the rewrites in Table 2.1 eliminate UB in all cases by performing exhaustive testing of 4-bit values, and we also formally verified a subset of the rewrites at their full widths (32 and 64 bits) using the Z3 theorem prover. The verification script is available online.[1]

The key invariant during this bottom-up process is that all subtrees of the node currently being analyzed are UB-free. Thus, when generation finishes, the entire expression is UB-free. This property can then be extended to other expressions by analyzing each expression in the order in which it will be evaluated during program execution. Conditionals do not present a problem for our approach because the analysis always knows which way a given conditional will go when the program executes.

This method for avoiding UB has several advantages over a more powerful backtracking scheme that could, for example, delete subtrees of expressions and regenerate them until UB-freedom is achieved. We implemented such a scheme and found that it usually requires multiple iterations before it succeeds and that sometimes it paints itself into a corner and becomes stuck; one of the reasons that this happens is that parameter shuffling (see Section 2.3.3.4) can significantly reduce the number of options available to YARPGen v.1. It would be possible to extend the backtracking scheme to get unstuck by backtracking further (and perhaps all the way back to the start of program generation), but the implementation would be more involved. In contrast, YARPGen v.1's rewrite rules are very efficient, never get stuck, and always succeed on the first try.

One might think that by generating programs whose behavior can be predicted statically, compiler testing would suffer because the programs would end up being too simple and would be mostly optimized away. This would only be the case if the compilers under test used link-time or whole-program optimization techniques. In other words, YARPGen v.1

[1] https://github.com/intel/yarpgen/blob/v1/rewrites.py

Table 2.1: UB elimination rewrite rules. When an unsafe condition holds, YARPGen v.1 avoids undefined behavior by replacing the problematic operation. MIN and MAX are INT_MIN and INT_MAX for operations performed on `int`-typed values, or LONG_MIN and LONG_MAX for operations performed on `long`-typed values. (Due to the "usual arithmetic conversions" in C [30] and C++ [30], mathematical operations are never performed on `char`- or `short`-typed values). MSB stands for the Most Significant Bit. The "signed or unsigned" criteria are more complicated for shifts because the operands to a shift are not converted to a common type.

Operation	**Unsafe condition**	**Signed or unsigned?**	**Replacement**
-a	a == MIN	S	+a
a + b	a + b > MAX || a + b < MIN	S	a - b
a - b	a - b > MAX || a - b < MIN	S	a + b
a * b	a * b > MAX || a * b < MIN , where a != MIN && b != -1	S	a / b
a * b	a == MIN && b == -1	S	a - b
a / b	b == 0	S or U	a * b
a / b	a == MIN && b == -1	S	a - b
a % b	b == 0	S or U	a * b
a % b	a == MIN && b == -1	S	a - b
a << b	MIN < b < 0	a is U && b is S	a << (b + c), where c ∈ [-b; -b + bit_width(a))
a << b	MIN < b < 0	a is S && b is S	a << (b + c), where c ∈ [-b; -b + bit_width(a) - MSB(a))
a << b	b == MIN	a is U or S && b is S	a
a << b	b >= bit_width(a)	a is U && b is U or S	a << (b - c), where c ∈ (b - bit_width(a); b]

Table 2.1 continued

Operation	**Unsafe condition**	**Signed or unsigned?**	**Replacement**
a << b	b>= bit_width(a)	a is S && b is U or S	a << (b - c), where c ∈ (b - bit_width(a) + MSB(a); b]
a >> b	MIN < b < 0	a is U or S && b is S	a >> (b + c), where c ∈ [-b; -b + bit_width(a))
a >> b	b == MIN	a is U or S && b is S	a
a >> b	b >= bit_width(a)	a is U or S && b is U or S	a >> (b - c) c ∈ (b - bit_width(a); b]
a >> b †	MIN < a < 0	a is S && b is U or S	(a + MAX) >> b
a >> b †	a == MIN	a is S && b is U or S	b

† implementation-defined behavior

relies on separate compilation to stop the compiler from propagating values between test initialization and test functions. If, in the future, compilers that we want to test make it difficult to disable link-time or whole-program optimizations, we may be forced to use a stronger optimization barrier, for example putting test code into a dynamically loaded library. We have tested this; it is trivial and does not require any changes to YARPGen v.1 itself, but rather only to the commands used to compile its output.

2.3.2.5 Lowering IR to the target language. When IR for the test functions is completely generated, it is lowered to either C or C++ and stored to a file. Lowering is a top-down recursive procedure. When lowering to C++, each array is randomly lowered to a C-style array, a `std::valarray`, a `std::vector`, or (for C++14 and newer) a `std::array`. There are also differences in array initialization across different versions of the C++ standard that must be taken into account.

2.3.2.6 Generating the test harness. Besides the file containing the test functions, YARPGen v.1 generates two additional files for each test case. First, a header containing declarations for all generated types, for the test functions, and for all global variables. Second, a driver file that defines the `main` function and defines (and optionally initializes) all global variables used by the test functions. The `main` function calls the test functions, computes a checksum by hashing values found in all output and mixed global variables (including struct fields and array elements), and then prints the checksum.

Given a particular set of implementation-defined behaviors, the C or C++ standard that is in effect completely determines the value of the checksum. In other words, all correct compilers, at all optimization levels, must generate executables that print the same checksum, which then serves as the key for differential testing. Any time changing the compiler or compiler options changes the checksum, we have found a compiler bug.

2.3.3 Generation Policies

When many random choices are drawn from the same distribution, randomly generated test cases can all end up looking somewhat alike. Our hypothesis was that this phenomenon reduces diversity in generated code and, in the long run, leads to less effective testing.

For YARPGen v.1, we developed *generation policies* that systematically skew probability distributions in ways that are intended to cause certain optimizations to fire more often. Some distribution changes are applied in randomly chosen sub-regions of the generated code. Since different generation policies typically compose gracefully, YARPGen v.1 allows them to overlap arbitrarily, leading to additional diversity in code generation. Other generation policies are applied to the entire test; in this case, to get diversity in tests, we rely on the fact that YARPGen v.1 will typically be run thousands or millions of times during a testing campaign.

These policies are drawn from our knowledge of how optimizing compilers work, and where bugs in them are likely to be found; it is not clear to us that this kind of insight can be automated in any meaningful fashion. We empirically evaluate the effectiveness of these generation policies in Section 2.4.

2.3.3.1 Arithmetic, logical, and bitwise contexts. An expression built from operators chosen from a uniform random distribution is less likely to contain dense clusters of operators like bitwise operators that would allow an optimization pass to perform rewrites, shown in Listing 2.1.

These examples are from comments in LLVM's `reassociate` pass[2] and instruction simplification analysis.[3] The first transformation is intended to create a situation where common subexpression elimination can further optimize the code; the second and third are standard peephole optimizations.

```
1  (x | c1) ^ (x | c2) --> (x & c3) ^ c3, where c3 = c1 ^ c2
2  (x & c1) ^ (x & c2) --> (x & (c1 ^ c2))
3  (x & ~y) | (x ^ y)  --> (x ^ y)
```

Listing 2.1: Examples of peephole optimizations that rely on dense clusters of bitwise operations.

[2] https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Transforms/Scalar/Reassociate.cpp#L1203-L1307

[3] https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Analysis/InstructionSimplify.cpp#L2138

YARPGen v.1 randomly chooses regions of code, or parts of expression trees, to use restricted subsets of mathematical operations, in order to more effectively trigger bugs in optimizations. These contexts are:

- additive: addition, subtraction, negation
- bitwise: binary negation, and, or, xor
- logical: and, or, not
- multiplicative: multiplication, division
- bitwise-shift: binary negation, and, or, xor, right/left shifts
- additive-multiplicative: addition, subtraction, negation, multiplication, division.

An example of expression built using the bitwise-shift context is shown in Listing 2.2.

2.3.3.2 Policies for constants. To ensure stress-testing of constant folding and related peephole optimizations, parts of expression trees are randomly selected to have all leaves as constants, or half of the leaves as constants. An example of code generated in the latter context is shown in Listing 2.3.

We found a bug in LLVM's bit-tracking dead code elimination pass (LLVM #33695) that we believe would have been very difficult to trigger without combining the shift-bitwise context with half of expression leaves being constants.

Since many peephole optimizations rely on special constants that are often small integers, YARPGen v.1 generates constants with small magnitude, and constants near values such as `INT_MAX`, more often than constants uniformly selected from the entire range. Transformations like the one shown in Listing 2.4 from LLVM's `instcombine` pass require constants

```
1  a = ~b & ((c | d) ^ (e << 4));
```

Listing 2.2: An example of expression generated in bitwise-shift context.

```
1  a = (((-2147483647 - 1) ^ b)
2       << (((-668224961 ^ 2147483647) + 1479258713) - 24)) |
3      (((c + 2147483647) >> 8) << 8);
```

Listing 2.3: An example of expression generated in the context where half of the leaves are constants.

```
(x + -1) - y --> ~y + x
```

Listing 2.4: An example of peephole optimization that relies on small constants.

of small magnitude, which would be very unlikely to occur if, for example, YARPGen v.1 selected constant values uniformly from $0..2^{64} - 1$.

Other constants are generated to have contiguous blocks of zeroes and ones in their binary representations.

Finally, YARPGen v.1 maintains a buffer of previously-used constants, and then in all but logical context randomly reuses these constants (and also their negations and complements) in subsequent parts of expression trees. This allows us to trigger transformations such as shown in Listing 2.5 more frequently.[4]

2.3.3.3 Common subexpression buffer. Common subexpression elimination, global value numbering, and related optimizations contain elaborate, bug-prone logic for factoring redundant computations out of code being compiled. To increase the chances of triggering these optimizations, YARPGen v.1 buffers previously-generated subexpressions and reuses them, as shown in Listing 2.6.

Code like this would be unlikely to be generated by a sequence of independent random choices.

2.3.3.4 Parameter shuffling. Each kind of random choice made by YARPGen v.1 is controlled by a distribution. To avoid the "every test case is different, but they all look the

```
(x | c2) + c --> (x | c2) ^ c2, iff (c2 == -c)
((x | y) & c1) | (y & c2) --> (x & c1) | y, iff c1 == ~c2
((x + y) & c1) | (x & c2) --> x + y, iff c2 = ~c1 and
                              c2 is 0b0...01...1 and (y & c2) == 0
```

Listing 2.5: Examples of peephole optimizations that rely on related constants.

[4] https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Transforms/InstCombine/InstCombineAddSub.cpp#L906
https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Transforms/InstCombine/InstCombineAndOrXor.cpp#L2538
https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Analysis/InstructionSimplify.cpp#L2208-L2211

```
1  d = c + (128 * a >> (b % 13));
2  e = INT_MAX - (128 * a >> (b % 13));
```

Listing 2.6: An example of code generated with common subexpressions.

same" phenomenon, YARPGen v.1 uses a parameter shuffling technique where, at startup time, random distributions are themselves randomly generated. So one test case might have 85% of its variables having `char` type, and the next might end up with only 3% of these. The insight behind shuffling is the same as the one behind swarm testing [15], but our implementation is finer-grained since it was designed into YARPGen v.1 instead of being retrofitted using command-line options. We shuffle the following controlling parameters:

- generation of compound types: base types for arrays, bit-field distribution in structs, whether to reuse existing compound types, etc.
- choices for arithmetic leaves: constant, variable, array element, structure member or pointer indirection
- where to put the results of assignments: local variable, mixed or output global variable, array element, struct member, pointer indirection
- array representation: C-style array, `std::valarray`, `std::vector`, `std::array`
- use of the subscript operator versus `at(i)` to access array elements
- arithmetic operators: unary, binary, ternary
- kind of statements: variable definition, assignment, conditional
- what kind of special constants to generate, how often we reuse existing constants and what modifications to them will we apply
- how many common sub-expressions we create and how often we reuse them
- distribution of contexts.

In total, there are 23 different parameters that are shuffled for each test. We believe this technique allows YARPGen v.1 to reach corner cases more effectively.

2.3.4 Automation

Figure 2.1 depicts the framework we developed for automating most of the important tasks associated with differential random compiler testing. This framework is not specific

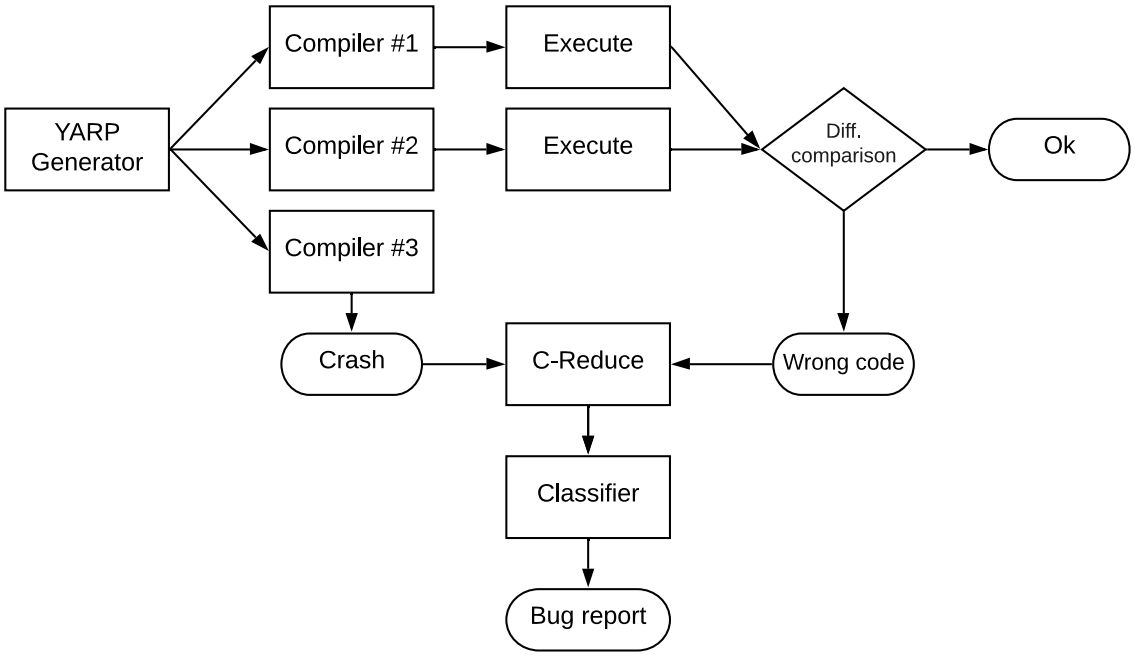

Figure 2.1: Random testing framework for differential random compiler testing.

to YARPGen v.1: we have also used it to run Csmith. Fewer than 10 lines of code needed to be changed to make this work.

On every core, the framework executes these steps in a loop:

1. invokes YARPGen v.1 to create a test case;
2. compiles the test using a configurable collection of compilers and compiler options, going to step (4) if any compiler crashes;
3. runs the resulting binaries, only continuing to the next step if behavioral differences in the output are detected;
4. generates an "interestingness" test script for C-Reduce [29], reduces the bug-triggering program, and
5. runs a bug classifier that we developed, in an effort to cluster together all test cases that trigger the same bug.

This framework uses a configurable number of cores and requires essentially no user intervention in the case where no compiler bugs are found. When bugs are found, they are ready to be reported to compiler developers because they are reduced and come with a Makefile for reproducing them. The automation framework optionally runs generated code

under UBSan, LLVM's undefined behavior sanitizer,[5] to check for bugs in the generator. It also takes care of cleaning up temporary files that are often left over when compilers crash, and supports optionally running failing tests multiple times, as a hedge against non-deterministic behavior (true non-determinism is rare in compilers, but we have seen incorrect bug reports because a test machine ran out of memory or disk space).

C-Reduce's transformations are syntactical in nature and tend to introduce undefined behavior during reduction. We rely on static and dynamic checking tools including compiler warnings, UBSan, ASan,[6] and tis-interpreter[7] to prevent undefined behaviors during test case reduction. Note that we do not rely on these tools to help YARPGen v.1 generate UB-free programs in the first place: there would be enormous overhead associated with invoking them incrementally on partially-constructed programs.

2.3.4.1 Classifying crashes and wrong-code bugs. The bug classifier is crucial because we observed that the number of times bugs are triggered typically follows a power law: one bug might be triggered by 1% of test cases, another by 0.01% of test cases, and yet another by 0.0001% of test cases. A recent study [31] appears to support this observation. It is important to prevent frequently-occurring bugs that have not been fixed yet from creating a needle-in-the-haystack phenomenon, hiding bugs that are triggered only very rarely.

Bugs that cause the compiler to crash are relatively easy to classify because they typically emit a characteristic error message. When a good error message is not emitted, for example, because the compiler was terminated by the OS due to a protection violation, we could attempt to classify the bug by looking at the compiler's stack trace, but we have not yet done this due to the limited engineering resources.

Bugs that cause the compiler to emit incorrect code are more difficult to classify, and here we require some cooperation from the compiler.[8] To do it, we search over the sequence of transformations the compiler performs while optimizing a program [32]. If leaving

---

[5] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[6] https://clang.llvm.org/docs/AddressSanitizer.html

[7] https://trust-in-soft.com/tis-interpreter

[8] https://llvm.org/docs/OptBisect.html

out a particular transformation makes the miscompilation disappear, then either that transformation is incorrect or else it ends up creating the conditions for a different, incorrect pass to introduce the miscompilation. Either way, this transformation is a useful fingerprint for the miscompilation, but it is not guaranteed to be a unique identifier for such errors.

2.3.5 Experimental Features

This section describes several features that we added to experimental versions of YARPGen v.1. We include them in our description because we believe them to be interesting and we used them to find compiler bugs. However, we wanted to describe them separately because they are, in general, less fleshed out and complete than are the features we describe elsewhere in this chapter. None of these features were used in evaluation experiments, described in Section 2.4.

2.3.5.1 Floating point. Generating code that manipulates floating point (FP) values is not difficult. The primary challenge is not in generation, but rather in interpreting the results of differential testing: it is not uncommon for a compiler optimization to change the result of a FP computation, and yet compiler developers do not consider this to be a bug. Our partial solution to this problem is to limit FP values to a narrow range (between 0.5 and 1.0), to limit the number of operations in an arithmetic chain containing FP values, and to exclude floating point values from YARPGen v.1's usual checksum mechanism. We instead dump the final values stored in FP variables to a file, and then perform differential testing using a threshold, rather than insisting on bit-for-bit equality. YARPGen v.1 supports the `float`, `double`, and `long double` types.

2.3.5.2 Vectors. The Clang C and C++ frontend for LLVM supports OpenCL vector extensions.[9] YARPGen v.1 has highly preliminary support for generation of short (2–16 element) vectors of the `int` and `unsigned` types. In order to be able to reuse YARPGen v.1's UB avoidance mechanisms, we forced each vector to contain the same value in all elements, and disabled accesses to individual elements. We also disabled logical, compari-

[9] https://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors

son, and ternary operators for vector-typed values because the OpenCL semantics for these operations diverges from C and C++.

2.3.5.3 Loops. Extending YARPGen v.1's machinery for avoiding undefined behaviors to support loops was an interesting challenge. Our solution uses a hybrid strategy. We test for the absence of undefined behavior in the first loop iteration and then ensure that the rest of the loop iterations operate on the same values as those seen in the first iteration. To make this happen, we disabled mixed variables (those that are both read and written), and we ensure that each array that is accessed inside a loop has the same value at all element positions. The loop iteration space is determined statically. Loop induction variables are treated specially: undefined behavior on them is avoided by construction, using the mechanism described in Section 2.3.1, and random accesses to induction variables from code in loop bodies are not allowed. Loop reduction variables (special-purpose variables used to, for example, sum up the values stored in an array) are treated in a similar fashion. Generated loops can contain regular conditional control flow, as well as the `break` and `continue` statements.

2.3.6 Limitations

YARPGen v.1 has a number of limitations including lack of support for pointer arithmetic, function calls, enums, classes, preprocessor definitions, templates, lambdas, dynamic memory allocation, compound assignment operations, and side-effecting operators. Additionally, support for floating point values and loops is quite limited. Many of YARPGen v.1's limitations—such as not generating code containing function calls—stem from its original goal of detecting bugs in scalar optimizations. Other limitations, such as not generating code containing strings or dynamic memory allocation, are implementation artifacts that we hope to lift in the future.

2.3.7 Implementation and Deployment

YARPGen v.1 is about 8,600 lines of C++. Its testing framework (written in about 3,100 lines of Python) is responsible for running tests across many cores, reducing programs that trigger bugs, and classifying bugs. Our software is open source; the version described

in this paper can be found at: https://github.com/intel/yarpgen/tree/v1 and the latest version can be found at: https://github.com/intel/yarpgen.

2.4 Evaluation

2.4.1 Summary of a Testing Campaign

Over a two-year period we used YARPGen v.1 to test the then-current versions of GCC, LLVM, and the Intel® C++ Compiler. In all cases for our main testing campaign, the compilers were generating code for various flavors of the x86-64 architecture. We also verified that YARPGen v.1 can be used to test compilers for ARM, but we didn't test it rigorously. Most of our testing was at the most commonly used optimization levels (e.g., `-O0`, `-O3`), but in some cases we also enabled non-standard compiler elements such as GCC's undefined behavior sanitizer[10] and LLVM's `NewGVN` pass.[11] The top-level results from this testing campaign are that we found

- 83 previously-unknown bugs in GCC, 78 of which have been fixed so far. 32 were compiler crashes, and 51 were wrong code bugs. 4 were in the front end, 50 were in the middle end, 11 were in the back end, and 18 of them are related to GCC's undefined behavior sanitizer. Some of these bugs are summarized in Table 2.2. The full list is available online.[12]
- 62 previously-unknown bugs in LLVM, 49 of which have been fixed so far. 53 bugs were compiler crashes, and 9 were wrong code bugs. One bug was in the front end, 15 were in the middle end, and 43 were in the back end (we could not categorize three bugs). Some of these bugs are summarized in Table 2.3. The full list is available online.[13] (This table contains one bug that the LLVM developers marked as DUPLICATE, and four that were marked as WORKSFORME. Usually, we would not count such bugs as valid, but in these five cases we verified that these were real, previously

[10] https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan/

[11] https://lists.llvm.org/pipermail/llvm-dev/2016-November/107110.html

[12] https://github.com/intel/yarpgen/blob/v1/gcc-bug-summary.md

[13] https://github.com/intel/yarpgen/blob/v1/llvm-bug-summary.md

Table 2.2: Reported GCC bugs. YARPGen v.1 found 83 GCC bugs. An “ICE” is an internal compiler error, or a crash. We shortened some names of components and bug descriptions.

#	ID	Status	Type	Component	Description
1	69820	fixed	wrong code	tree-opt	Test miscompiled with -O3 option
2	70021	fixed	wrong code	target	Test miscompiled with -O3 option for -march=core-avx2.
3	70026	fixed	ICE	tree-opt	ICE in expand_expr_real_2 with -O1 -ftree-vectorize
4	70153	fixed	ICE	c++	ICE on valid C++ code
5	70222	fixed	wrong code	rtl-opt	Test miscompiled with -O1
6	70251	fixed	wrong code	tree-opt	Wrong code with -O3 -march=skylake-avx512.
7	70252	fixed	ICE	tree-opt	ICE in vect_get_vec_def_for_stmt_copy with -O3 -march=skylake-avx512.
8	70333	fixed	wrong code	target	Test miscompiled with -O0.
9	70354	fixed	wrong code	tree-opt	Wrong code with -O3 -march=broadwell and -march=skylake-avx512.
10	70429	fixed	wrong code	rtl-opt	Wrong code with -O1.
11	70450	fixed	wrong code	target	Wrong code with -O0 and -O1.
12	70542	fixed	wrong code	rtl-opt	Wrong code with -O3 -mavx2.
13	70725	fixed	ICE	tree-opt	ICE on valid code
14	70726	fixed	ICE	tree-opt	ICE on valid code
15	70728	fixed	wrong code	target	GCC trunk emits invalid assembly for knl target
16	70902	suspended	ICE	rtl-opt	GCC freezes while compiling for ’skylake-avx512’ target
17	70941	fixed	wrong code	target	Test miscompiled with -O2.
18	71259	fixed	wrong code	tree-opt	GCC trunk emits wrong code

Table 2.2 continued

#	ID	Status	Type	Component	Description
19	71261	fixed	ICE	tree-opt	Trunk GCC hangs on knl and broadwell targets
20	71279	fixed	ICE	middle-end	ICE on trunk gcc with knl target
21	71281	fixed	ICE	target	ICE on gcc trunk on knl, wsm, ivb and bdw targets (tree-ssa-reassoc)
22	71389	fixed	ICE	target	ICE on trunk gcc on ivybridge target (df_refs_verify)
23	71470	fixed	wrong code	target	Wrong code on trunk gcc with westmere target
24	71488	fixed	wrong code	tree-opt	Wrong code for vector comparisons with ivybridge and westmere targets
25	71655	fixed	ICE	tree-opt	GCC trunk ICE on westmere target
26	71657	new	wrong code	target	Wrong code on trunk gcc (std::out_of_range), westmere
27	72835	fixed	wrong code	tree-opt	Incorrect arithmetic optimization involving bitfield arguments
28	73714	fixed	wrong code	tree-opt	Incorrect unsigned long long arithmetic optimization
29	77476	fixed	wrong code	target	[AVX-512] illegal kmovb instruction on KNL
30	77544	fixed	ICE	tree-opt	segfault at -O0 - infinite loop in simplification
31	78132	fixed	wrong code	rtl-opt	GCC produces invalid instruction (kmovd and kmovq) for KNL.
32	78436	fixed	wrong code	tree-opt	incorrect write to larger-than-type bitfield (signed char x:9)
33	78438	fixed	wrong code	rtl-opt	incorrect comparison optimization
34	78720	fixed	wrong code	tree-opt	Illegal instruction in generated code
35	78726	fixed	wrong code	middle-end	Incorrect unsigned arithmetic optimization
36	79399	fixed	ICE	middle-end	GCC fails to compile big source at -O0
37	80054	fixed	ICE	tree-opt	ICE in verify_ssa with -O3 -march=broadwell/skylake-avx512

Table 2.2 continued

#	ID	Status	Type	Component	Description
38	80067	fixed	ICE	sanitizer	ICE in fold_comparison with -fsanitize=undefined
39	80072	fixed	ICE	tree-opt	ICE in gimple_build_assign_1 with -O3 -march=broadwell/skylake-avx512
40	80297	fixed	ICE	c++	Compiler time crash: type mismatch in binary expression
41	80341	fixed	wrong code	middle-end	gcc miscompiles division of signed char
42	80348	fixed	ICE	sanitizer	UBSAN: ICE in ubsan_instrument_division
43	80349	fixed	ICE	sanitizer	UBSAN: ICE with "type mismatch in binary expression" message
44	80350	fixed	wrong code	sanitizer	UBSAN changes code semantics when -fno-sanitize-recover=undefined is used
45	80362	fixed	wrong code	middle-end	gcc miscompiles arithmetic with signed char
46	80386	fixed	wrong code	sanitizer	UBSAN: false positive - constant folding and reassosiation before instrumentation
47	80403	fixed	ICE	sanitizer	UBSAN: ICE with "type mismatch in binary expression" message in / and % expr
48	80536	fixed	ICE	sanitizer	UBSAN: compile time segfault
49	80597	fixed	ICE	ipa	ICE: in compute_inline_parameters, at ipa-inline-analysis.c:3126
50	80620	fixed	wrong code	tree-opt	gcc produces wrong code with -O3
51	80800	fixed	wrong code	sanitizer	UBSAN: yet another false positive
52	80875	fixed	ICE	sanitizer	UBSAN: ICE in fold_binary_loc at fold-const.c:9817
53	80932	fixed	wrong code	sanitizer	UBSAN: false positive as a result of distribution: c1*(c2*v1-c3*v2)=>c1*c2*v1-c1*c3*v2
54	81065	fixed	wrong code	sanitizer	UBSAN: false positive as a result of distribution involving different types
55	81088	fixed	wrong code	middle-end	UBSAN: false positive as a result of reassosiation
56	81097	fixed	wrong code	sanitizer	UBSAN: false positive for not existing negation operator and a bogus message

Table 2.2 continued

#	ID	Status	Type	Component	Description
57	81148	fixed	wrong code	sanitizer	UBSAN: two more false positives
58	81162	fixed	wrong code	tree-opt	UBSAN switch triggers incorrect optimization in SLSR
59	81281	fixed	wrong code	sanitizer	UBSAN: false positive, dropped promotion to long type.
60	81387	unconfirmed	wrong code	sanitizer	UBSAN consumes too much memory at -O2
61	81403	fixed	wrong code	tree-opt	wrong code at -O3
62	81423	fixed	wrong code	rtl-opt	Wrong code at -O2
63	81488	fixed	wrong code	sanitizer	gcc goes off the limits allocating memory in gimple-ssa-strength-reduction.c
64	81503	fixed	wrong code	tree-opt	Wrong code at -O2
65	81546	fixed	ICE	tree-opt	ICE at -O3 during GIMPLE pass dom
66	81553	fixed	ICE	rtl-opt	ICE in immed_wide_int_const, at emit-rtl.c:607
67	81555	fixed	wrong code	tree-opt	Wrong code at -O1
68	81556	fixed	wrong code	tree-opt	Wrong code at -O2
69	81588	fixed	wrong code	tree-opt	Wrong code at -O2
70	81607	fixed	ICE	c++	Conditional operator: "type mismatch in shift expression" error
71	81705	fixed	wrong code	middle-end	UBSAN: yet another false positive
72	81814	fixed	wrong code	middle-end	Incorrect behaviour at -O0 (conditional operator)
73	81987	fixed	ICE	tree-opt	ICE in verify_ssa with -O3 -march=skylake-avx512
74	82192	fixed	wrong code	rtl-opt	gcc produces incorrect code with -O2 and bit-field
75	82353	fixed	wrong code	sanitizer	runtime ubsan crash

Table 2.2 continued

#	ID	Status	Type	Component	Description
76	82381	fixed	ICE	tree-opt	ICE: qsort checking failed
77	82413	fixed	ICE	c	-O0 crash (ICE in decompose, at tree.h:5179)
78	82576	new	ICE	rtl-opt	sbitmap_vector_alloc() not ready for 64 bits
79	82778	fixed	ICE	rtl-opt	crash: insn does not satisfy its constraints
80	83221	fixed	ICE	tree-opt	qsort comparator not anti-commutative: -2147483648, -2147483648
81	83252	fixed	wrong code	target	Wrong code with "-march=skylake-avx512 -O3"
82	83382	new	wrong code	sanitizer	UBSAN tiggers false-positive warning [-Werror=uninitialized]
83	83383	fixed	wrong code	tree-opt	Wrong code with a bunch of type conversion and ternary operators

Table 2.3: Reported LLVM bugs. YARPGen v.1 found 62 LLVM bugs. An “ICE” is an internal compiler error, or a crash. We shortened some names of components and bug descriptions.

#	ID	Status	Type	Component	Description
1	25517	worksforme	ICE	llc	[AVX-512] llc generates incorrect code
2	25518	worksforme	ICE	llc	[AVX-512] llc generates incorrect code
3	25519	worksforme	ICE	llc	[AVX-512] llc generates incorrect code
4	27584	fixed	ICE	llc	ICE with knl target
5	27591	fixed	ICE	Backend: X86	ICE with KNL target, Assertion ‘Emitted && "Failed to emit a zext!"’ failed
6	27638	fixed	ICE	Backend: X86	ICE in llvm::SDValue llvm::X86TargetLowering::LowerSETCC
7	27789	fixed	ICE	llc	ICE on knl target
8	27873	fixed	ICE	New Bugs	ICE in llvm::TargetLowering::SimplifyDemandedBits on knl
9	27879	fixed	ICE	Frontend	ICE on trunk llvm (Invalid operands for select instruction)
10	27997	fixed	ICE	New Bugs	ICE on trunk Clang, knl target, Assertion ‘L.isLCSSAForm(DT)’ failed
11	28119	fixed	ICE	llc	[AVX-512] llc UNREACHABLE executed at lib/IR/ValueTypes.cpp:128
12	28291	fixed	ICE	New Bugs	ICE with knl target (Assertion ‘C1->getType() == C2->getType())
13	28301	fixed	ICE	New Bugs	Assertion "Register is not used by this instruction! failed
14	28312	fixed	ICE	New Bugs	ICE with knl target (Assertion ‘Res.getValueType() == N->getValueType(0))
15	28313	fixed	ICE	New Bugs	ICE with knl target (Assertion ‘isSCEVable(V->getType()))
16	28661	fixed	wrong code	New Bugs	[AVX512] incorrect code for boolean expression at -O0.
17	28845	new	ICE	New Bugs	Incorrect codegen for "store <2 x i48>" triggered by -fslp-vectorize-aggressive
18	29058	fixed	ICE	New Bugs	Assert in llvm::SelectionDAG::Legalize()

Table 2.3 continued

#	ID	Status	Type	Component	Description
19	30256	fixed	ICE	New Bugs	Assert in llvm::ReassociatePass::OptimizeAdd
20	30286	fixed	wrong code	New Bugs	[AVX-512] KNL bug at -O0
21	30693	fixed	ICE	Backend: X86	AVX512 Segfault: alignment incorrect for relative addressing
22	30775	fixed	ICE	Backend: X86	Assertion "NodeToMatch was removed partway through selection"′ failed.
23	30777	worksforme	ICE	New Bugs	Assertion "Illegal cast to vector (wrong type or size)"′ failed
24	30783	fixed	ICE	New Bugs	Assertion "replaceAllUses of value with new value of different type!"′ failed
25	30813	fixed	ICE	New Bugs	Assertion ‘ShiftBits != 0 && MaskBits <= Size’ failed.
26	31044	new	ICE	Backend: X86	Assertion "Expect to be extending 32-bit registers for use in LEA"′ failed
27	31045	fixed	ICE	Backend: X86	Clang fails in insertDAGNode
28	31306	fixed	ICE	Backend: X86	[AVX-512] ICE: Cannot select: t41: v8i64 = X86ISD::SUBV_BROADCAST
29	32241	fixed	wrong code	New Bugs	Incorrect result with -march=skx -O0 -m32.
30	32256	fixed	ICE	New Bugs	Assertion "Cannot move between mask and h-reg"′ failed with -m32 -O0
31	32284	fixed	ICE	New Bugs	Assertion "Invalid child # of SDNode!"′ failed with -O0 -march=skx.
32	32316	fixed	ICE	New Bugs	Assertion "Binary operator types must match!"′ failed with -O1 -march=skx
33	32329	fixed	ICE	New Bugs	Silent failure in X86 DAG->DAG Instruction Selection with -march=skx -O3
34	32340	fixed	ICE	Common CodeGen	Assertion "Deleted Node added to Worklist"′ failed with -O0
35	32345	fixed	ICE	Common CodeGen	Assertion "Mask size mismatches value type size!"′ failed with -O0.
36	32515	fixed	ICE	Common CodeGen	Assertion "DELETED_NODE in CSEMap!" fires a lot with -march=skx
37	32525	duplicate	ICE	Backend: X86	Assertion "Number of nodes scheduled doesn’t match the expected number!"

Table 2.3 continued

#	ID	Status	Type	Component	Description
38	32830	fixed	wrong code	New Bugs	Clang produces incorrect code with -O2 and higher
39	32894	fixed	ICE	Backend: X86	ICE "Cannot replace uses of with self" in llvm::SelectionDAG
40	33442	new	wrong code	Frontend	UBSAN: right shift by negative value is not caught for some values
41	33560	fixed	ICE	Backend: X86	Cannot emit physreg copy instruction at lib/Target/X86/X86InstrInfo.cpp:6707
42	33695	fixed	wrong code	New Bugs	Bit-Tracking DCE fails to invalidate nsw/nuw after transforms
43	33765	fixed	ICE	New Bugs	Replacing instructions in instcombine violates dominance relation
44	33828	fixed	ICE	Common CodeGen	Assertion "getConstant with a uint64_t value that doesn't fit in the type!" failed
45	33844	fixed	ICE	Backend: X86	Assertion 'loBit <= BitWidth && "loBit out of range"' failed with -O1
46	34137	fixed	ICE	Backend: X86	clang crash in llvm::SelectionDAG::Combine on -O0
47	34377	new	wrong code	Backend: X86	Clang produces incorrect code with -O1 and higher
48	34381	fixed	wrong code	New Bugs	Clang produces incorrect code at -O0
49	34782	fixed	ICE	New Bugs	Assertion "Deleted edge still exists in the CFG!"
50	34787	fixed	ICE	New Bugs	Assertion 'TN->getNumChildren() == 0 && "Not a tree leaf"'
51	34828	fixed	ICE	Register Allocator	Assertion "index out of bounds!"' failed
52	34830	new	ICE	New Bugs	Clang failing with "Cannot emit physreg copy instruction"
53	34837	fixed	ICE	New Bugs	UNREACHABLE in DAGTypeLegalizer::PromoteIntegerResult
54	34838	fixed	ICE	New Bugs	InstCombine - Assertion "Compare known true or false was not folded"
55	34841	fixed	ICE	Scalar Opt	InstCombine - Assertion "Cannot convert from scalar to/from vector"
56	34855	fixed	ICE	Backend: X86	Cannot select shl and srl for v2i64

Table 2.3 continued

#	ID	Status	Type	Component	Description
57	34856	fixed	ICE	New Bugs	Assertion on vector code: "Invalid constantexpr cast!"
58	34934	new	ICE	Backend: X86	UNREACHABLE at LegalizeDAG.cpp: "Unexpected request for libcall!"
59	34947	fixed	ICE	Backend: X86	-O0 crash: assertion "Cannot BITCAST between types of different sizes!"
60	34959	fixed	wrong code	New Bugs	Incorrect predication in SKX, opt-bisect blames SLP Vectorizer
61	35272	fixed	ICE	New Bugs	Assertion "Invalid sext node, dst <src!" in llvm::SelectionDAG::getNode()
62	35583	fixed	ICE	New Bugs	NewGVN indeterministic crash

unknown bugs first reported by us. In other words, the compiler developers labeled these bugs incorrectly.)

- 76 previously-unknown bugs in the Intel® C++ Compiler, 67 of which have been fixed so far. 30 bugs were compiler crashes, and 46 bugs were wrong code. 35 were in scalar optimizations, 23 were in the back end, 17 were in the high-level optimization and vectorization framework, and one was in inter-procedural optimizations.

Taken together, these results show that YARPGen v.1 is capable of finding defects in production-grade compilers that have already been heavily targeted by random test-case generators. Furthermore, most of these defects were deemed worth fixing by compiler developers.

2.4.1.1 Examples of reduced, bug-triggering programs. Listing 2.7 demonstrates the advantages of explicit common subexpression generation and special constants, while Listing 2.8 shows the advantage of contexts.

2.4.1.2 Concurrently reported bugs. One way to gain confidence that a fuzzer is finding bugs that matter to real developers is to look for cases where a bug found by the fuzzer

```
1  #include <stdio.h>
2
3  unsigned short a = 41461;
4  unsigned short b = 3419;
5  int c = 0;
6
7  void foo() {
8    if (a + b * ~(0 != 5))
9      c = -~(b * ~(0 != 5)) + 2147483647;
10 }
11
12 int main() {
13   foo();
14   printf("%d\n", c);
15   return 0;
16 }
```

Listing 2.7: GCC bug #81503, found on revision #250367. The correct result is `2147476810`; the incorrectly optimized version prints `-2147483648`.

```
//func.c:
extern const long int a;
extern const long int b;
extern const long long int c;
extern long int d;

void foo() {
    d = ((c | b) ^ ~c) & (a | ~c);
}

//////////////////////////////////////////////////

//main.c:
#include <stdio.h>

extern void foo ();

const long int a = 6664195969892517546L;
const long int b = 1679594420099092216L;
const long long int c = 1515497576132836640LL;
long int d = -7475619439701949757L;

int main () {
    foo ();
    printf ("%ld\n", d);
    return 0;
}
```

Listing 2.8: LLVM bug #32830, found on revision #301477. The correct result is `-236191838907956185`; the incorrectly optimized version prints `-5292815780869864331`.

was independently re-reported by an application developer. About 10% of the LLVM bugs that we reported were independently discovered by projects such as SQLite (LLVM #35583), Chromium (LLVM #34830 and #33560), LibreOffice (LLVM #32830), Speex (LLVM #32515), and GROMACS (LLVM #30693). For GCC, only one bug was independently discovered, in Python 3.5.1 (GCC #71279).

This situation usually happened in two different cases. In the first case, we and application developers who test their product against the latest compiler version reported the same bug around the same time (usually within a couple of days from each other). In the second

case, a bug that we had reported remained unfixed for some time (sometimes more than a year), and then, later on, application developers rediscovered it in the released compiler.

Finding a compiler bug using a fuzzer, rather than waiting for application developers to find it, is preferable for two important reasons. First, whereas our YARPGen v.1-based workflow is directed entirely towards finding compiler bugs and has significant machinery aimed at making this task easier, finding a compiler bug that causes a large application to be miscompiled is extremely non-trivial, even for experienced developers. Second, test-case reduction for programs generated by YARPGen v.1 is automated and works very well. In contrast, extracting a minimal reproducer for a compiler bug from, for example, the Chromium sources is laborious. In practice, the reduced test cases that we report are much smaller than those reported by application developers.

2.4.1.3 Partial bug fixes. YARPGen v.1 is good at finding *partial bug fixes*. These occur when a compiler developer, for one reason or another, eliminates the symptoms of a compiler bug without fully addressing the root cause. This leads to the situation where the test case in the bug report no longer triggers the bug, but other test cases can still trigger it. Fuzzers seem to be particularly good at finding different ways to trigger errors, providing thorough testing of bug fixes. During our bug-finding campaign we discovered five LLVM bugs (#34381, #33695, #32284, #32256, #30775) and six GCC bugs (#83252, #81588, #80932, #80348, #71488, #70333) that were partially fixed.

2.4.1.4 Sanitizer bugs. YARPGen v.1's testing system optionally uses the undefined behavior sanitizer supported by GCC and LLVM to ensure that generated tests are free from undefined behavior. Because YARPGen v.1 was designed to avoid all undefined behaviors, every alarm signaled by a sanitizer either indicates an error in YARPGen v.1 or in the sanitizer. We found two bugs in LLVM and 22 bugs in GCC that were in the undefined behavior sanitizer, or else were exposed due to using the sanitizer.

2.4.2 Evaluating the Direct Impact of Generation Policies

A major hypothesis behind YARPGen v.1 was that generation policies would make it more effective at finding compiler bugs. We conducted an experiment with the goal of

confirming or rejecting this hypothesis. Our starting point was the set of LLVM crash bugs found by YARPGen v.1 which could be reliably identified by a specific error message printed by LLVM as it terminated. 22 bugs met this criterion; the rest of them had non-specific signatures, such as segmentation faults. We did not look at miscompilation bugs in this experiment because it is difficult to reliably determine that a particular miscompilation bug has been triggered (our bug classifier is useful, but not foolproof). All of the experimental features described in Section 2.3.5 were disabled for this experiment.

Out of the 22 bugs, we could not trigger 13 of them within 72 hours when using all threads on a 16-core (32 thread) Intel® Xeon® machine, using the latest version of YARPGen v.1. We dropped these from the experiment—we deemed them impractical for our experiment due to their low probability of being triggered. For the remaining nine bugs, we ran YARPGen v.1 for 72 hours (on the same kind of Intel® Xeon® machine) in its standard configuration—where it uses generation policies—and also for 72 hours in a configuration where it does not use generation policies.

The results of this experiment are summarized in Table 2.4. Four of the bugs were not triggered at all without generation policies. Out of the remaining five bugs, two were triggered more often without generation policies and three were triggered more often with generation policies. We also used a zero-mean test to see if the data support rejecting the null hypothesis "generation policies do not improve YARPGen v.1's ability to trigger each bug" with 95% confidence. In summary, in four cases generation policies allowed a bug to

Table 2.4: Bug rediscovery results. Number of times nine selected LLVM crash bugs were found in 72 hours on a 32-thread machine, with and without generation policies (GP).

Bug ID	GP	No GP	GP found the bug more times?	GP is better at 95%?
27638	57029	67976	no	no
27873	23	2	yes	yes
29058	9	0	yes	yes
30256	21	0	yes	yes
30775	1	0	yes	no
32284	153	21	yes	yes
32316	5843	27922	no	no
32525	1	0	yes	no
33560	1853	1720	yes	yes

be found that could not be found otherwise, whereas the reverse situation (a bug could be found without generation policies, but not with them) never occurred.

2.4.3 Evaluating an Indirect Impact of Generation Policies

Getting good statistical evidence about real compiler bugs is difficult, so we also investigated a less direct metric: the impact of generation policies on the number of times different optimizations were triggered, as measured using optimization counters supported by LLVM and GCC. An optimization counter simply records the number of times a particular optimization, such as common subexpression elimination, is used during a compilation. The basis for this experiment is that we believe (but have not experimentally confirmed) that triggering an optimization more times is better, because this will be more likely to uncover incorrect corner cases in the optimization. Optimization counters can be viewed as a kind of domain-specific code coverage metric: they are incremented at locations in the compiler code where experts believe the resulting information will be interesting or useful. For example, they are incremented when analysis can draw some conclusion or when optimization transformation is successfully applied. All of the experimental features described in Section 2.3.5 were disabled for this experiment.

We conducted an experiment where we ran YARPGen v.1, with and without generation policies, for 30 repetitions of 48 minutes each (for 24 hours of total testing time) on a 32 core (64 thread) AMD Ryzen™ Threadripper machine. We performed this experiment separately for GCC and for LLVM, monitoring a collection of optimization counters that we had identified ahead of time as being the kind that we intend YARPGen v.1 to trigger often. (Other optimization counters, such as those that measure loop or floating point optimizations, would not have been relevant to this experiment.) We then used a zero-mean test to put each counter into one of three categories. First, those where we can be 95% confident that generation policies trigger the counter more times (i.e., generation policies improve testing capability). Second, those where we can be 95% confident that generation policies trigger the counter fewer times (i.e., generation policies worsen testing capability). Finally, a default category where neither of these conditions is true, corresponding to a null

hypothesis where generation policies do not decisively make testing better or worse. The results are summarized in Table 2.5.

For testing GCC, generation policies have a clear advantage (the benefit of using them outweighs disadvantages), whereas the LLVM results are mixed. We took a closer look at the counters in `instcombine` (LLVM's large collection of peephole optimizations). Many of these showed an increase, and those that decreased did not do so catastrophically. These results are shown in Table 2.6.

Finally, for every optimization counter, we computed the ratio between the counter values with and without generation policies. The geometric mean of the resulting ratios is 1.2 for LLVM and 1.4 for GCC. This shows that overall, generation policies increase the number of optimizations triggered. The important thing, however, isn't a uniform increase but rather diversity across a large number of test cases. Therefore, even though generation policies decrease the number of times some optimizations are triggered, our

Table 2.5: Effect of generation policies (GP) on optimization counters.

	Number of LLVM counters	Number of GCC counters
GP is better	94	67
GP is worse	108	27
Comparable	20	11
Total	222	105

Table 2.6: Effect of generation policies on selected optimization counters. Effect of generation policies (GP) on values of several LLVM optimization counters. "GP is better" means that a 95% confidence test was passed.

Counter	GP is better	Ratio of GP to no GP
instcombine.NumCombined	yes	1.04
instcombine.NumConstProp	no	0.78
instcombine.NumDeadInst	no	0.97
instcombine.NumDeadStore	yes	1.2
instcombine.NumExpand	yes	7.4
instcombine.NumFactor	yes	1.8
instcombine.NumReassoc	yes	1.8
instcombine.NumSel	yes	155.0
instcombine.NumSunkInst	no	0.95
instsimplify.NumExpand	yes	3.7
instsimplify.NumReassoc	yes	2.4
instsimplify.NumSimplified	no	0.76

goals can be met by simply turning off generation policies for some fraction of YARPGen v.1's runs, ensuring that optimizations that end up being suppressed by generation policies get triggered enough times.

2.4.4 Performance of YARPGen v.1

Table 2.7 shows where CPU time is spent during a differential testing run. For this experiment we spent 24 hours testing GCC and LLVM, each at their `-O0` and `-O3` optimization levels, on a machine with two 28-core (56-thread) Intel® Xeon® Platinum 8180 processors. We used YARPGen v.1 in its standard, out-of-the-box configuration, with the experimental features described in Section 2.3.5 disabled. We measured CPU consumption from our Python test harness, and consequently we did not measure how much CPU time the harness itself used. However, since the test harness performs almost no real computation, we do not believe that it contributed significantly. Overall, YARPGen v.1 accounted for less than 5% of the total CPU time; almost all of the rest of the CPU time was consumed by the compilers, which are particularly CPU-hungry when they are asked to optimize heavily.

2.4.5 YARPGen v.1 and Code Coverage

The last experiment we performed was to investigate what effect compiling programs generated by YARPGen v.1 had on code coverage of the open-source compilers. For both GCC and LLVM, we collected code coverage information from all combinations of

1. the suite of unit tests that come with the compiler,

Table 2.7: Distribution of CPU time used during fuzzing. Distribution of CPU time used during 24 hours of differential testing of GCC and LLVM, each at two optimization levels, on a large multicore system.

tool	step	% of total CPU time
YARPGen v.1	generation	4.98%
gcc -O0	compilation	19.54%
	execution	0.03%
gcc -O3	compilation	28.96%
	execution	0.03%
clang -O0	compilation	14.95%
	execution	0.03%
clang -O3	compilation	31.43%
	execution	0.03%

2. the C and C++ programs from SPEC CPU 2017, compiled at `-O3`, and
3. random testing using YARPGen v.1 in its default configuration, invoking the compiler at `-O3`.

For both compilers, we collected coverage information using a configure-time option provided by the compilers' respective build systems. All of the experimental YARPGen v.1 features described in Section 2.3.5 were disabled for this experiment.

Tables 2.8 and 2.9 summarize the results. A "region" is a unit of coverage that is similar to a basic block: multiple lines of C or C++ that do not contain control flow can belong to the same region. However, it is also possible for a single line of code containing internal control flow, such as `return x || y && z`, to contain multiple regions.[14]

Table 2.8: Coverage of LLVM source code.

	Functions	Lines	Regions
YARPGen v.1	28.53%	33.71%	24.78%
SPEC CPU 2017	36.14%	41.90%	33.68%
SPEC + YARPGen v.1	36.57%	42.48%	34.34%
% change	+0.43%	+0.58%	+0.66%
unit test suite	84.33%	87.61%	78.87%
unit tests + YARPGen v.1	84.34%	87.72%	79.04%
% change	+0.01%	+0.11%	+0.17%
unit tests + SPEC	84.36%	87.79%	79.15%
unit tests + SPEC + YARPGen v.1	84.37%	87.83%	79.21%
% change	+0.01%	+0.04%	+0.06%

Table 2.9: Coverage of GCC source code.

	Functions	Lines	Branches
YARPGen v.1	40.70%	34.41%	26.20%
SPEC CPU 2017	52.15%	45.18%	34.23%
SPEC + YARPGen v.1	52.82%	47.39%	37.06%
% change	+0.67%	+2.21%	+2.83%
unit test suite	80.22%	78.28%	62.82%
unit tests + YARPGen v.1	80.38%	79.31%	64.41%
% change	+0.16%	+1.03%	+1.59%
unit tests + SPEC	80.31%	78.59%	63.28%
unit tests + SPEC + YARPGen v.1	80.44%	79.46%	64.65%
% change	+0.13%	+0.87%	+1.37%

[14] https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#interpreting-reports

Regardless of the coverage metric, it is clear that random testing using YARPGen v.1 does not improve coverage very much. (Our finding is compatible with numbers reported in Table 3 of the Csmith paper [6]). Similarly, the compilers are covered somewhat poorly by compiling SPEC CPU 2017. We believe that a reasonable takeaway from these results is that none of function, line, or region coverage is very good at capturing the kind of stress testing that is generated by a random program generator. It seems possible that a path-based coverage metric would capture this better.

2.5 Conclusion

YARPGen v.1 is effective at finding bugs in modern C and C++ compilers, even when they have been heavily tested by other random test-case generators. Over a two-year period, we found and reported 83 previously unreported bugs in GCC, 62 in LLVM, and 76 in the Intel® C++ Compiler. Many of these bugs have been fixed. Some of the bugs we found were independently rediscovered by the developers of large software projects, confirming that at least some of the bugs triggered by YARPGen v.1 also affect applications' code. Moreover, in cases where we reported a bug that was also reported by application developers, the reduced test cases in our bug reports were considerably smaller.

Our first main research contribution is generation policies: a mechanism for improving diversity in random programs. Generation policies also increase the number of times targeted optimizations are performed by LLVM by 20%, and by 40% for GCC, during a given amount of time devoted to random testing, increasing the likelihood that optimization bugs will be triggered. Another contribution is a mechanism for avoiding undefined behavior for relatively complex code including control flow, pointers, and arrays. Finally, YARPGen v.1 comes with a testing framework that automates essentially all tasks related to random compiler testing except for actually reporting the bugs.

CHAPTER 3

FUZZING LOOP OPTIMIZATIONS

Adapted from V. Livinskii, D. Babokin, and J. Regehr, "Fuzzing loop optimizations in compilers for C++ and data-parallel languages," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Art. no. 181, Jun. 2023. DOI: 10.1145/3591295.

3.1 Introduction

Machine learning, big data, and other recent trends have caused loop-specific compiler optimizations to increase in importance and in sophistication. For example, both GCC and LLVM can use the polyhedral model [34, 35] to analyze and transform loop nests. LLVM's polyhedral optimizer—its "polly" subproject[1]—contains 24,700 lines of C++, and relies on another 175,000 lines of external support code. This much code, performing tricky symbolic reasoning, written in an unsafe language, and engineered to run quickly, might be expected to contain bugs and, in fact, a recent study of defects in GCC and LLVM [36] found that "loop optimizations in both GCC and LLVM are more bug-prone than other optimizations." When a bug causes the compiler to crash, it is annoying; when it causes the compiler to emit incorrect object code, it is potentially dangerous. For instance, Listing 3.1 shows a loop optimization bug in GCC that we discovered and reported.

The research problem addressed by this chapter is how to expose bugs in the implementations of sophisticated loop optimizers. The main hypothesis that we evaluate is that loop optimization bugs in compilers for C-like data parallel languages can be effectively discovered using randomly generated programs that are free of undefined behavior (UB), and that also contain idioms recognized by loop optimizers. Randomized testing is desirable because the space of inputs to a compiler is very large and, empirically, compiler developers are not

[1] https://polly.llvm.org

```
void test(unsigned short a, unsigned short b, long long c) {
  for (char i = 0; i < (char)c; i += 5) {
    if (!b)
      var_120 = a;
    else
      var_123 = a;
  }
}
```

Listing 3.1: This function triggers a loop-related miscompilation bug in GCC (#102920); it was automatically reduced from a test case that we generated.

capable of writing test cases by hand that reveal all of the defects in a production-grade compiler. We require generated programs to be UB-free because it is not possible to reach reliable conclusions from the observable behaviors of a program that executes UB.

Although random generation of UB-free code is a problem that has been addressed by a number of previous papers, we are unaware of prior work specifically focusing on generating the kinds of expressive loop idioms that appear to be required to stress-test loop optimizers. Listing 3.2 shows an example of the kind of code that we want to generate; it is a stencil: a loop where each element of an output array is a function of a collection of nearby elements of an input array or arrays. In this case, each output element is the average of three neighboring input elements. To optimize this code, a compiler can notice that `in[i + 1]` in one loop iteration is the same value as `in[i]` in the next iteration, and also `in[i - 1]` in the iteration after that. Thus, while a naive translation of this loop would load all three values from RAM each time the loop body executes, an optimized version only needs

```
void stencil(int* restrict in, int* restrict out, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = (in[i - 1] + in[i] + in[i + 1]) / 3;
}

```

Listing 3.2: Even a very simple stencil loop like this triggers interesting optimizations that we want to test.[2]

[2] https://gcc.godbolt.org/z/aEs1daYWq

to load one value from RAM, with the other two values being forwarded from previous iterations using registers.

Modern compilers for C and C++ use sophisticated loop transformations and auto-vectorization to achieve high performance, while data-parallel languages such as ISPC [37] and SYCL [38] require less aggressive analysis and optimization since the languages directly expose fine-grained parallelism. However, compilers for all of these languages perform non-trivial code transformations, which are error-prone, especially when targeting modern CPU and GPU architectures. Our work targets all of these C-family languages by generating random programs in a high-level intermediate representation that supports loop idioms and also static analysis to ensure UB-freedom; it can be lowered relatively straightforwardly to the four different concrete syntaxes. As a starting point, we used our previous version of YARPGen [25] (described in Chapter 2), which was designed to target scalar optimizations in C and C++ compilers, but we almost entirely re-implemented it to support loops and to output multiple languages. In this chapter, we will refer to the old implementation as YARPGen v.1, and to the new one as YARPGen v.2. The version that we started with contained 8,619 lines of C++; to patch that program (using Git's patch facility) to the current YARPGen v.2 requires removing 6,295 lines of C++ while adding 10,099.

YARPGen v.2 was able to detect 66 previously-unknown bugs in GCC, 28 in LLVM, 16 in the Intel® oneAPI DPC++ compiler, and 12 in Intel® ISPC. Furthermore, although these targets were not a primary focus for us, we found two bugs in the Intel® Software Development Emulator[3] and two in the Alive2 translation validation tool [39]. We reported all of these bugs, and most of them have been fixed, showing that compiler developers consider the kind of bugs that we find to be worthwhile. The research contributions of this dissertation include a static UB avoidance mechanism for loops and our loop generation policy mechanism for creating programs that helps us find interesting and difficult-to-trigger compiler bugs.

[3] https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html

3.2 Background: Underspecified Aspects of C-Family Programming Languages

To support generating high-quality object code across a wide variety of target platforms, the C language and its descendants are somewhat underspecified: they give implementations substantial freedom to make convenient and efficient choices. These choices come in three flavors.

First, *implementation-defined behaviors* are those where the compiler must make a consistent choice and also document it. For example, the size of the `int` type, and the range of values that it supports, are implementation-defined. These behaviors do not concern us in this work. In principle, they limit the scope of differential testing, which can only be done across compilers that agree on a collection of important implementation-defined behaviors. However, in practice, most modern compilers of interest agree on these.

Second, *unspecified behaviors* are those where there compiler must choose from a collection of alternatives, with no requirement for consistency. For example, the order in which arguments to a function are evaluated is unspecified. In practice, these behaviors are few enough and benign enough that they can be avoided fairly easily—for example, by ensuring that function arguments do not have side effects.

Finally, *undefined behaviors* in C-family languages are untrapped error conditions: the standard imposes no requirement on the behavior of a program that, for example, accesses out-of-bounds memory. There are hundreds of undefined behaviors (UBs), and it is, in general, difficult to statically guarantee their absence. Avoiding UB while generating expressive loops is a primary contribution of our work.

3.3 Generating UB-Free Programs with Expressive Loops

Our code generator's output should be expressive: it should be syntactically and semantically interesting in the sense that it triggers as many code paths in the compiler (and, in particular, the loop optimizer) as possible. On the other hand, its output must not execute undefined behaviors, and ideally it should avoid UB without using the kind of pervasive

dynamic checking that, for example, Csmith [6] used. These goals are in tension; this section describes how we achieve them both. Figure 3.1 provides an overview. Generation proceeds in three main steps: creating a high-level program skeleton, fleshing out the skeleton with details such as arrays and operations on them, and then lowering our intermediate representation (IR) to a concrete syntax.

3.3.1 Test Oracles

Software testing requires an *oracle* to determine if some execution of the system under test was correct or incorrect. In addition to the trivial oracle that looks for abnormal termination of the compiler process, YARPGen v.2 supports:

- *Ground truth:* As a side effect of undefined behavior avoidance, YARPGen v.2 precomputes the effect of executing the randomly generated code. Thus, a miscompilation can be signaled if the result of running the compiled code differs from this prediction.
- *Differential:* We compute a checksum of the effect of executing the randomly generated code; this is used for differential testing where the result is not known in advance, but a bug is signaled when two different compilers (or two different modes of the same compiler, such as `gcc -O0` and `gcc -O3`) disagree with each other.

Both oracles are necessary in practice. The first one is useful when differential testing is impossible, for example, because there is only one compiler for a given language. This is the case for Intel® ISPC, where we have found bugs that affect all modes of the compiler, such as issue #1768; differential testing is incapable of finding this bug. However, precomputing the result of a program interferes with automated test-case reduction tools (e.g., C-Vise [40], C-Reduce [29], and Perses [41]) because they make non-semantics-preserving changes during reduction. Thus, it is crucial that we do not rely on the code producing a specific answer—this is where differential testing becomes most useful. (This sort of test-case reduction relies on external tools to reject reduction steps that trigger undefined behaviors.)

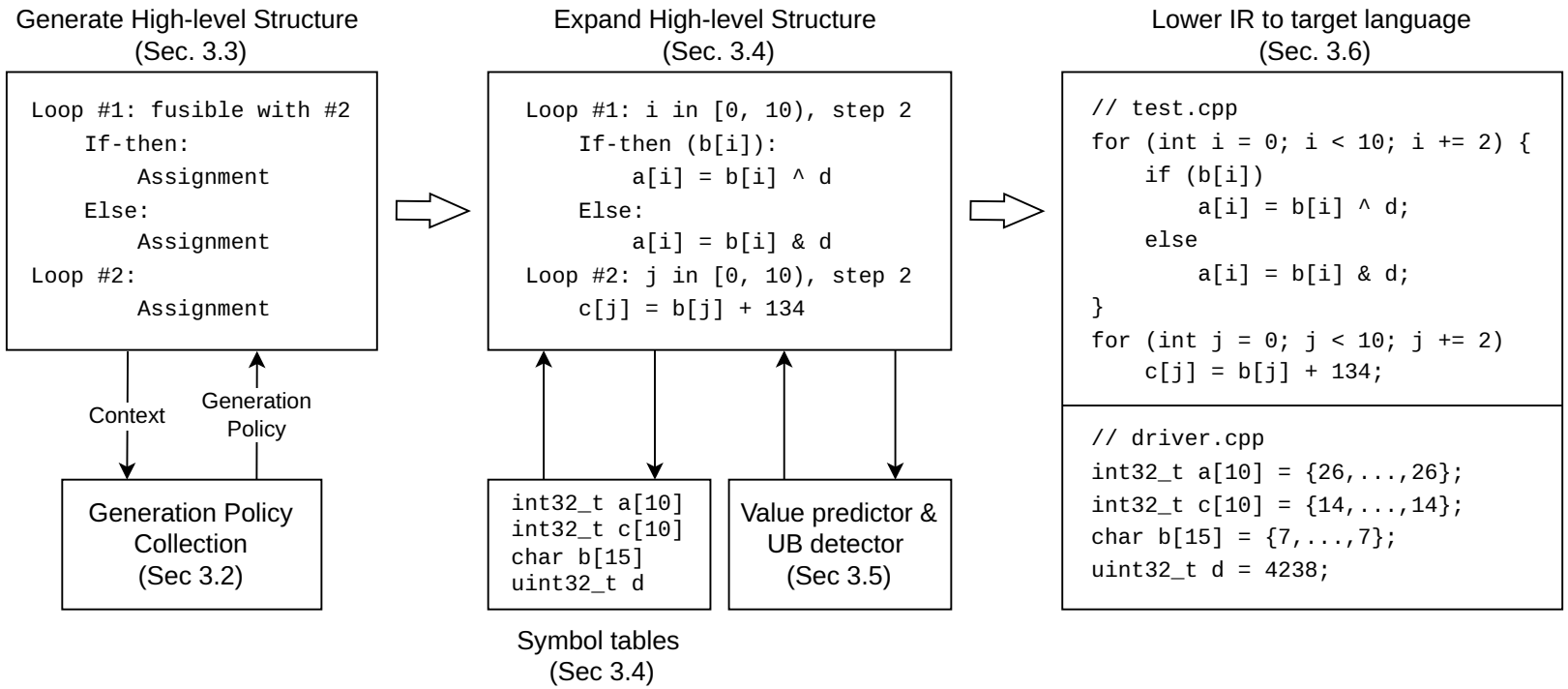

Figure 3.1: Overview of how YARPGen v.2 generates code.

3.3.2 Loop Generation Policies

Generation policies is an idea introduced in Section 2.3.3. We used it as something of an umbrella term for mechanisms that were used to increase the probability of generating test cases that were believed to be desirable, and that (without generation policies) were being generated only rarely, if at all. In this chapter, we have adapted this idea, broadly construed, to generating loops. As a simple example, consider that we would like to test *loop fusion*, an optimization that can merge adjacent loops when they have identical iteration spaces and also lack dependencies that block fusion. The odds of a pair of fusible loops being organically generated is quite low, but it is easy to generate such a pair simply by making the decision to do so.

The basic principle behind generation policies is that we cannot find bugs in optimizations that we cannot trigger. A rule of thumb that we have used is that if we run across an interesting loop transformation that is supported by more than one compiler, then we should at least consider creating a generation policy that targets it. We used existing bugs reports, unit tests, test suites, and general knowledge of common compiler optimizations as sources of inspiration. We analyzed these to identify interesting patterns that were missing from YARPGen v.2 at the time and tried to fit them into the existing infrastructure or extend it to support them. A difficult issue is helping YARPGen v.2 trigger specific transformations without constraining expressiveness to such an extent that fuzzing effectiveness is compromised. What we attempt to do is abstract away the essence of the pattern that triggers the transformation, while leaving to random chance as many of the details of instantiating the pattern as possible. The resulting fuzzing technique ends up being something like a gray-box fuzzer, but with a human—a YARPGen v.2 developer—in the loop, adding generation policies as required in response to gaps in code coverage. We have added loop generation policies to YARPGen v.2 piecemeal over a period of several years as we learned how to break loop optimizers; the rest of this section describes them. They are not mutually exclusive; they compose to increase the diversity of the generated code even further. For example, YARPGen v.2 can generate a combination of reduction and stencil (`a += (b[i - 1] + b[i] + b[i - 1]) / 3`) that will compute a reduction of three elements of the array.

3.3.2.1 Loop sequences and loop nests. These are represented as a first-class elements in the fuzzer's IR; they are the main factor that determines the high-level shape of the generated code. They are essential for making coordinated decisions involving multiple loops. For instance, to trigger a loop interchange optimization, we have to generate a loop nest that contains array accesses with a column-major order. To trigger loop fusion, adjacent loops have to have the same iteration spaces. To trigger other loop optimizations, "perfectly nested loops" where all assignment operators are in the innermost loop must be generated.

3.3.2.2 Array access patterns. Applications often access arrays in idiomatic patterns [42], and optimizing compilers have adapted to provide specific support for some of these. These patterns are determined by the interaction between arrays and loops' iteration spaces. First, we can arbitrarily decide the relation between array dimensionality and the loop nest depths (fewer, same, or more). Second, we can pick the order of the induction variables used in the array subscripts—in order of the loop nest or not, and whether to use the same induction variable to access multiple array dimensions.

This approach allows us to achieve good expressiveness and to mimic various access patterns that are found in real applications. For example, if we decide that the array dimension matches the loop depth and use the same induction variables for all dimensions (`a[i][i]`), we will get a diagonal traversal of a matrix. Another option is to use in-order traversal of an array with some constant indexes to get a slice. This generation policy causes all of these special cases to happen regularly. We consider array access patterns to be a set of rules that govern generation of individual array subscripts—subsequent policies such as stencils make more involved decisions that are synchronized across multiple array subscripts.

3.3.2.3 Stencils. Stencil codes (Listing 3.2) are ubiquitous in image processing and scientific applications, such as finite difference methods. Given the huge number of degrees of freedom available to a naive random program generator, it is not likely that stencils will be generated naturally. Therefore, we implemented a stencil pattern as a first-class IR element that can be used in arithmetic expressions inside loops. Our stencil generation policy results in loops that use multiple constant offsets from a single induction variable into an array or

arrays. This gives us direct, fine-grained control over things like stencil size, stride, number of array dimensions used, and number of arrays used. Otherwise, it would be difficult to get control over these elements if we only used unstructured random generation.

3.3.2.4 Vectorizable loops. Automated vectorization is a sophisticated program transformation performed by most of the compilers we are targeting. It tends to be somewhat fragile, and can be defeated by a number of program properties such as:

- data dependencies across loop iterations
- library functions outside of a limited set
- unpredictable iteration spaces
- unpredictable control flow, particularly related to exiting the loop.

Furthermore, it is often the case that only the innermost loop of a loop nest gets vectorized. If we want to heavily stress autovectorizers, we need to ensure that all prerequisites are met sufficiently often. In our initial experiments, this did not happen naturally, so we added a "vectorizable" loop property that ensures that these prerequisites be fulfilled, satisfying our goal of generating many vectorizable loops, but not compromising on our ability to express more general loops.

3.3.2.5 Reductions. These computations—common in real applications—reduce the dimensionality of data, for example by summing the elements of a loop, computing the smallest element, etc. YARPGen v.2's reduction generation policy provides a generalized version of this kind of computation where a randomly generated function is applied element-wise to an input array, resulting in an output array of reduced dimensionality.

3.3.2.6 Loops over bytes. Loops that iterate over a vector of bytes are common, and optimizing compilers like to turn them into more efficient computations when possible. For instance, open-coded loops over bytes can sometimes be turned into faster implementations such as the system-provided `memset` or `memcpy` routine (Listing 3.3). These idiom recognizers tend to be fragile, and they are highly desirable targets for stress testing. To achieve this, YARPGen v.2 employs a byte-loop pattern by creating a loop header that iterates through bytes, and populates the loop body with randomly generated expressions.

```
1 extern int a[], b[];
2 void foo() {
3   for (int i = 0; i < 20; ++i)
4     a[i] = b[i];
5 }
```

Listing 3.3: Code snippet that triggers LLVM's `memcpy` idiom recognizer.

3.3.2.7 Compiler-specific loop attributes. It can be useful to override compiler's built-in cost functions using pragmas such as LLVM's `#clang vectorize`, `#clang interleave`, and `#clang unroll`. We optionally add them to loops when generate tests for C and C++. It is safe to do so, because when a compiler encounters unknown pragmas, it simply emits warning messages.

3.3.3 Generating the High-Level Program Structure

Because our focus is on intra-procedural loop optimizations, YARPGen v.2 generates a single function when it runs. Its first step, structure generation, performs top-down construction of skeleton code that it will later flesh out with details. The high-level code includes elements found in C-family languages such as conditionals and assignments, but it also contains larger structural elements such as loop sequences and loop nests. During this step, generation policies are invoked to determine properties of various program elements. For example, generation policies supply the maximum depth of a loop nest and the maximum length of a loop sequence. Other generation-policy-based decisions are also made during this phase, such as a loop sequence being marked as containing loops that have a common iteration space, or an individual loop being marked as auto-vectorizable. These attributes will be used later to guide how YARPGen v.2 fills in the detailed code. The reason that we first create the high-level skeleton is that this led to a pleasing separation of concerns in the random generator, making it more modular and debuggable than it otherwise would have been. For instance, as a debugging aid, YARPGen v.2 can print a representation of the high-level structure, which can be inspected readily because it lacks complicating details.

3.3.4 Expanding the Skeleton

The first step in fleshing out the skeleton program is to generate global data items that the random code will access. Data is split into a table of inputs, whose values are known and will not be changed by the randomly generated code, and outputs, whose values will be inspected in order to detect miscompilation bugs. Since YARPGen v.2 does not attempt to look for bugs in floating point optimizations, these globals are integer-typed scalars and arrays. On the input side, only scalar variables are created in this step, with arrays being created later, on the fly, in conjunction with loop generation, to ensure that array sizes match up with loop iteration spaces. Output variables and arrays are created later during the expansion of assignments. If any of the global variables end up being unused by the randomly-generated code, they are removed before IR is lowered to a concrete syntax. The values stored in these global variables are known to YARPGen v.2—which uses them for its undefined behavior analysis—but are opaque to the compiler that is being tested. We currently rely on separate compilation to provide opacity; if we ever test a compiler that is sufficiently aggressive with cross-file optimization, we will have to be heavier handed. For example, the initialized globals could be placed in a dynamically loaded library.

After global input variables have been created, YARPGen v.2 proceeds to fill in the IR for the generated code. Expression nodes in the IR are expanded top-down; the targets of assignments are allocated in the output symbol table, exposing the resulting values to external compiler-correctness-checking. Each scope gets its own symbol table; these local tables contain iterators in addition to scalars and arrays. Iterators are treated separately by the undefined behavior analysis: they are used as induction variables in loops, but they are not involved in arbitrary randomly generated computations. Local symbol tables store the type, dimension, scope, initial value, and current value of each data item.

To expand a loop header, YARPGen v.2 first has to take into account constraints imposed by high-level loop properties that come from generation policies. For example, a loop that is part of a fusible sequence will have the same iteration space as its neighboring loops. A loop performing a stencil computation will likely have an iterator that does not (quite) run over the entire array, because the stencil will access array elements offset from the iterator's position. Initially, each loop is characterized by start, end, and step values that are

constants, but YARPGen v.2 randomly replaces some of these constants with expressions that are partially opaque to the compiler (by depending on external variables), but end up evaluating to the same values that the constants would have held. This gives us some interesting diversity in how loop iteration spaces look to the compiler without making UB avoidance more complicated than it already is.

3.3.5 Static Undefined Behavior Avoidance for Loops

Loops potentially lead to two kinds of UB: out-of-bounds array accesses and arithmetic UB inside the loop body. Whereas Csmith [6] relied on dynamic checks to avoid both kinds of UB, our hypothesis is that pervasive conditional control flow in loop bodies would hamper some of the optimizations that we wish to test. Existing research [10, 11] appears to corroborate this suspicion.

YARPGen v.2's undefined behavior avoidance is entirely static, and uses concrete value tracking, which was pioneered by the Orange family of random program generators [12–14] and extended in Section 2.3.1. For example, when generating a shift operator in a C-family language, the shift exponent must be non-negative and also smaller than the bitwidth of the value being shifted. Thus, if YARPGen v.1 wanted to generate `x << y` in a situation where `y == -1000`, it would instead generate something like `x << (y + 1005)`. As long as the code being generated is loop-free, this strategy maintains the invariant that the already-generated program fragment is UB-free. Thus, when the tool terminates, the entire program is UB-free. For scalar code this approach worked well, and YARPGen v.2 also uses it. This UB avoidance strategy is not altogether straightforward to extend to loops, though both Orange and YARPGen v.1 had some limited solutions for this (Section 2.3.5). Their approaches are summarized in Chapter 6, so here we will give a brief overview. Orange3 simulated every loop iteration, avoiding UB any time that it would have happened by adding values from an array specifically initialized with values that avoid UB. This approach is computationally expensive, and can produce UB avoidance artifacts in the generated code, similar to Csmith's [6] wrapper functions. Orange4 [43] simply ensured that each generated loop would execute at most once. This approach is not able to test loop

optimizations that rely on run-time properties of the code, such as loop trip count for some case of unrolling or vectorization. YARPGen v.1 had experimental support for loops that, like Orange4, ensured that the first iteration would not trigger UB; but then it also ensured that subsequent iterations of the loop would see the same values.

YARPGen v.2 uses this same approach—ensuring that all loop iterations see the same values—as one of its two approaches to generate UB-free loop code. This is possible because we keep track of every iteration space that we generate, as well as dimensions of the arrays, so that these can be matched up. It also helps that we maintain a clean separation between variables that are only used as inputs, and those that are only used as outputs, making value tracking pretty straightforward even inside loop bodies. We rely on separate compilation (Section 3.3.4) to ensure that the compiler cannot notice that we have used these strategies, forcing it to analyze the general case.

Separate compilation cannot, however, stop the compiler from observing runtime characteristics of our code. For example, at runtime, ISPC maintains an execution mask to track active program instances[4] in order to use masked vector instructions to describe control flow, rather than using explicit branching. For instance, the `max` function (Listing 3.4) is compiled into a single instruction. On hardware targets that do not support masking directly, the cost of applying a mask is high, and it becomes profitable to check for the "all-on" state of the mask at runtime, and execute a separate, simpler code path in that case. Our first method produces loops that operate on the same values in each iteration. Therefore, the mask will always contain the same value, and we will only test the "all-on" code path and

```
1  varying int max(varying int a, varying int b) {
2      return a > b ? a : b;
3  }
```

Listing 3.4: ISPC code for a parallel `max` operation; `varying` indicates a vector data type. This function is compiled into a single masked vector instruction: `vpmaxsd zmm0, zmm0, zmm1`.[5]

[4] An ISPC program instance is similar to a CUDA thread or an OpenCL work-item https://ispc.github.io/ispc.html#basic-concepts-program-instances-and-gangs-of-program-instances

[5] https://ispc.godbolt.org/z/MraWhd5Tb

will never trigger the general path. We would instead like the execution mask to contain diverse values so all code paths based on it are tested.

To ensure that we can test this sort of optimization, we developed a novel generation mechanism that allows arrays to use multiple values (but without attempting to analyze all loop iterations, which is computationally infeasible). We do this by logically separating the iteration space of the loop into subsets. At present, we support partitioning into two subsets—even and odd elements—but we plan to expand and generalize this support in the future. In this scheme, the iterator walks through the array in an alternating odd-even pattern, so the loop body performs a different computation for the even and odd values. For example, if we start at element one and use three as a step, we will get the required pattern.

We reuse the first method (same values in each iteration) to generate loop body expressions that use arrays with multiple values. First, we create an expression for one of the subsets (we arbitrarily choose the even subset as the primary one). Next, we check if the newly generated expression also does not contain UB for the odd subset. If it does not, we successfully generated an expression that does not contain UB for both subsets, so we add it to the test. Otherwise, we eliminate UB separately for each subset and use versioning code to combine the resulting expressions, as shown in Line 2 of Listing 3.5. Optionally, YARPGen v.2 can hide the splitting criterion from the compiler, as shown in Line 3 of Listing 3.5, using an opaque variable.

The resulting minimal runtime diversity is sufficient to test optimizations that use it to select between different versions of optimized code. Such optimizations usually use a simple lattice (mask is all-on, all-off, or mixed) to make a decision about the execution path.

```
1 for (int i = 0; i < N; ++i) {
2     a[i] = (i % 2 == 0)    ? (b[i] + c[i]) : (b[i] - c[i]);
3     d[i] = (i % 2 == zero) ? (e[i] * f[i]) : (e[i] / f[i]);
4 }
```

Listing 3.5: Addition of two arrays, where UB is avoided by conditional access, based on the iterator. In the second case, the condition is obfuscated with a variable whose value is opaque to the compiler.

Therefore, a combination of a simple odd-even pattern and uniform values is sufficient to test all code paths, created by such optimizations.

A limitation of this method is that arrays can have different values only along one of the axes. For example, in the case of a three-dimensional array, alternating values can be in rows, columns, or planes, but only in one of these for any given loop nest.

The main advantage of loop partitioning is that it permits YARPGen v.2 to reuse its static UB-avoidance mechanism in loop bodies with minimal overhead in terms of time taken to generate random code: the UB avoidance mechanism runs twice per loop instead of once.

The iteration space partitioning method has several advantages over the previous approaches. It allows us to test divergent values in the loop body without any wrapper functions, it does not create UB avoidance artifacts, and it only requires us to analyze a small, fixed number of loop iterations, minimizing overhead. This approach is one of the two major contributions of this chapter.

3.3.6 Lowering YARPGen v.2 IR to Multiple Languages

Because the four C-family languages supported by YARPGen v.2 are fairly similar, and because its IR has been designed taking all of them into account, lowering our intermediate representation to any of them is reasonably straightforward. One useful thing that we can do during lowering, when the target language supports several similar constructs, is to choose from them randomly. For example, when targeting ISPC and SYCL, we randomly choose between emitting regular and data-parallel loops, driving the compiler down different code paths.

The two most similar of the supported languages are C and C++. From the YARPGen v.2 point of view, they are almost identical. The difference between them includes different UB rules for the left-shift operator, supported types, and standard library functions. The rest of the machinery is shared.

Support for ISPC, on the other hand, is more involved. The main complexity comes from its explicit vector types, which means that YARPGen v.2 has to support an additional type parameter to capture this in its IR. This type property is orthogonal to C++ type

properties, which add another layer of type casting rules. They have to be supported in the form of explicit casts, as well as implicit casting rules that are similar to integral promotion and arithmetic conversions in C++. We also extend support for standard library calls to reductions and vector-wise operations.

YARPGen v.2's SYCL support is limited and did not require any augmentation of the fuzzer IR. We were able to satisfy the limitations of parallel SYCL loops by changing generation parameters, such as maximum depth of data-parallel loops or set of allowed standard library functions. The main modifications were implemented in the lowering component of the fuzzer, where we had to support data transfer between the test driver and test function.

3.3.7 High-Level Intermediate Representation

The high-level IR in YARPGen v.2 is designed to support expressive, UB-free loops, and also to lower to multiple target languages. This IR is roughly analogous to a compiler IR, in the sense that our IR nodes represent program objects such as types, values, statements, and expressions. It is, overall, simpler than a compiler's IR because YARPGen v.2 does not need to support the wide variety of analyses and transformations that an optimizing compiler supports. Rather than discovering properties of something like the iteration space of a loop, YARPGen v.2 tends to make a decision about the iteration space ahead of time and then subsequently generates code having the desired properties.

Our IR supports generation of expressive, UB-free loops in two ways. First, we carefully chose what elements have a first-class representation in the IR; these include loop sequences, loop nests, and stencils. Compiler IRs, in contrast, would usually represent these elements implicitly as collections of more primitive nodes. By directly representing higher-level abstractions, we can more easily enforce high-level properties such as creating perfect loop nests or synchronizing the iteration spaces of a sequence of loops. The second way that our IR supports expressiveness and UB-freedom is by supporting a variety of auxiliary elements that track the program environment and the current state of the generation process. For example, generating a loop that can be reliably vectorized requires restricting its behavior in several ways (Section 3.3.2); the loop context object helps track these.

Supporting lowering to multiple programming languages is mainly a matter of avoiding commitment to particular representations too early. For instance, matrix multiplication in ISPC uses data-parallel loops with carefully constructed operations to avoid data-dependency conflicts, whereas C++ uses simple loop nests. These code fragments look quite different at the syntax level, but since they perform the same operation, we represent them using the same IR construct. Similarly, vector types in ISPC serve some of the same functions as arrays in other languages; we have IR elements that abstract over this representation issue.

3.3.8 Generating a Test Harness

Besides a file containing the randomly generated function, YARPGen v.2 also emits a header file containing declarations for global variables and a driver file that contains a main function and also definitions for all global variables. The randomly generated code typically receives some of its inputs via parameters and others via global variables. The main function initializes data items, calls the randomly generated function, and then looks at its return value and also the values of global variables; to support the differential testing oracle, it prints a checksum of these outputs, and to support the ground truth oracle, it checks for mismatches with its value predictions.

3.3.9 Limitations

Features not yet supported by YARPGen v.2 include:

- Floating-point math
- Dynamic memory allocation
- Support for multiple random functions—generated code includes function calls, but only to standard library code
- First-class pointers and pointer arithmetic—YARPGen v.2 currently only supports the limited kinds of pointers that occur when an array value decays into a pointer type
- Non-standard vector extensions, such as intrinsic functions that give C or C++ code access to specific vector instructions.

Some of these limitations are inherent to our design. For example, we did not set out to find bugs in vector intrinsics, and in fact, optimizing compilers more or less leave these intrinsics alone during compilation, treating them as opaque. Fixing other limitations is clearly desirable and is a matter of putting more engineering resources into YARPGen v.2. For instance, supporting multiple functions that call each other should be fairly straightforward given the infrastructure that we have already created. Finally, some limitations seem difficult to lift. The most prominent example is the absence of floating-point (FP) support. This is a serious limitation, but it is on par with the state of the art in C++ compiler fuzzing. To the best of our knowledge, there exist only two compiler fuzzers (Orange [14] and YARPGen v.1 [25]) that have addressed the issue of correct compilation of IEEE FP, and both of them have serious limitations and did not appear to generate significant results.

3.4 Evaluation

This section describes the bugs that have been found using YARPGen v.2 and evaluates its ability to trigger various loop optimizations. These results—including the number of reported bugs—are completely separate from our previous work on YARPGen v.1, described in Chapter 2.

3.4.1 Summary of a Testing Campaign

Over the three-year period, we used YARPGen v.2 to test then-current versions of GCC and LLVM, as well as occasionally testing Intel® ISPC, the Intel® oneAPI DPC++ compiler, and Alive2. An up-to-date list of bugs found by YARPGen v.2, which we have reported, is available online.[6] In all cases, compilers targeted various flavors of x86-64. We checked that YARPGen v.2 can be used to test ARM programs (using an Apple M1 chip), but we did not perform a thorough testing campaign for that target. Our testing focused on commonly used optimization levels: `-O0` and `-O3`. It had been shown that esoteric compiler options can increase the number of detected bugs [44]. However, we avoided these since, in our experience, compiler developers are often not motivated to fix issues that are encountered far away from the default optimization pipeline.

[6] https://github.com/intel/yarpgen/blob/main/bugs.rst

The top-level results of our testing campaign are

- 66 bugs in GCC. All of these were either fixed or assigned to compiler developers, showing that bugs discovered by YARPGen v.2 are valued by the GCC developers. 32 of these were miscompilation bugs, 31 were compiler crashes, and three were cases where the compiler failed to terminate. 56 of the bugs were in the middle-end optimizations, seven in the x86-64 back end, two were in an inter-procedural optimization, and one was in a front end. 22 of the bugs we reported affected more than one released version of the compiler. 13 of the bugs that we reported were independently rediscovered by users. Table 3.1 shows the 50 most recently reported bugs.
- 28 bugs in LLVM. 18 of these were fixed, one was confirmed, one was resolved, and eight remain unacknowledged. 23 of these bugs were compiler crashes and five were miscompilation bugs. 15 were in a middle-end optimization, 11 in the x86-64 back end, and two were not classified. A full list is in Table 3.2.
- 12 bugs in Intel® ISPC. All of these have been fixed. Seven of them were compiler crashes and five were miscompilation bugs. Seven were in a middle-end optimization and five in the x86-64 back end. A full list is in Table 3.3.
- 16 bugs in the Intel® oneAPI DPC++ compiler. Nine were miscompilation bugs and seven were compiler crashes. Nine were in a middle-end optimizations, three in the x86-64 back end, and four remain unclassified. Intel® oneAPI DPC++ bugs were reported to a non-public bug tracker.
- Two bugs in Intel® SDE and two bugs in Alive2.

3.4.1.1 Overview of reported bugs. We looked for patterns in the bugs that we found. Please note that since our testing campaign ran over several years, during which we continued to develop YARPGen v.2, features that we added earlier were used in more test cases—so there are likely to be some biases in these informal results.

In terms of the overall program structure, we determined that perfect loop nests of depth two with unknown trip count were the most common bug trigger. We have reported bugs with bigger loop depths (up to five), but their combined presence was roughly the

Table 3.1: Reported GCC bugs. YARPGen v.2 found 66 bugs in GCC. “ICE” stands for “Internal Compiler Error.” “ASGD” stands for “Assigned.” We shortened names of the components and bug descriptions.

#	ID	Status	Type	Component	Description
1	91137	fixed	wrong code	tree-opt	Wrong code with -O3
2	91145	fixed	ICE	tree-opt	ICE: in vect_build_slp_tree_2, at tree-vect-slp.c:1143
3	91178	fixed	ICE	tree-opt	Infinite recursion in split_constant_offset in slp after r260289
4	91204	fixed	ICE	target	ICE in expand_expr_real_2, at expr.c:9215 with -O3
5	91207	fixed	wrong code	tree-opt	Wrong code with -O3
6	91293	fixed	wrong code	tree-opt	Wrong code with -O3 -mavx2
7	91403	assigned	ICE	tree-opt	GCC fails with ICE.
8	94727	fixed	wrong code	tree-opt	GCC produces incorrect code with -O3
9	95248	fixed	wrong code	tree-opt	GCC produces incorrect code with -O3 for loops
10	95268	fixed	ICE	tree-opt	ICE: invalid ‘PHI’ argument
11	95284	fixed	ICE	tree-opt	ICE: verify_gimple failed
12	95295	fixed	wrong code	tree-opt	g++ produces incorrect code with -O3 for loops
13	95297	fixed	ICE	tree-opt	ICE: Segmentation fault
14	95308	fixed	ICE	tree-opt	ICE: in maybe_canonicalize_mem_ref_addr with -O3 -march=skylake-avx512
15	95401	fixed	wrong code	tree-opt	GCC produces incorrect instruction with -O3 for AVX2
16	95487	fixed	ICE	tree-opt	ICE: verify_gimple failed (error: invalid vector types in nop conversion)
17	95649	fixed	ICE	tree-opt	ICE during GIMPLE pass: cunroll
18	95717	fixed	ICE	tree-opt	ICE during GIMPLE pass: vect: verify_ssa failed

Table 3.1 continued

#	ID	Status	Type	Component	Description
19	95916	fixed	ICE	tree-opt	ICE during GIMPLE pass: slp : verify_ssa failed
20	96022	fixed	ICE	tree-opt	ICE during GIMPLE pass: slp in operator[], at vec.h:867
21	96755	fixed	ICE	target	ICE in final_scan_insn_1, at final.c:3073 with -O3 -march=skylake-avx512
22	98048	fixed	ICE	tree-opt	ICE in build_vector_from_val, at tree.c:1985
23	98064	fixed	ICE	tree-opt	ICE in check_loop_closed_ssa_def, at tree-ssa-loop-manip.c:726 with -O3
24	98069	fixed	wrong code	tree-opt	Miscompilation with -O3
25	98211	fixed	wrong code	tree-opt	Wrong code at -O3
26	98213	fixed	timeout	tree-opt	Never ending compilation at -O3
27	98302	fixed	wrong code	target	Wrong code on aarch64
28	98308	fixed	ICE	tree-opt	ICE in vect_slp_analyze_node_operations, at tree-vect-slp.c:3764
29	98381	fixed	wrong code	tree-opt	Wrong code with -O3 -march=skylake-avx512
30	98513	fixed	wrong code	tree-opt	Wrong code with -O3
31	98640	fixed	wrong code	tree-opt	GCC produces incorrect code with -O1 and higher
32	98694	fixed	wrong code	target	GCC produces incorrect code for loops with -O3 for skx and icx
33	99777	fixed	ICE	tree-opt	ICE in build2, at tree.c:4869 with -O3
34	99927	fixed	wrong code	rtl-opt	Wrong code
35	100081	fixed	timeout	tree-opt	Compile time hog in irange
36	101014	fixed	timeout	tree-opt	Big compile time hog with -O3
37	101256	fixed	wrong code	tree-opt	Wrong code with -O3

Table 3.1 continued

#	ID	Status	Type	Component	Description
38	102511	fixed	wrong code	tree-opt	GCC produces incorrect code for -O3: first element of the array is skipped
39	102572	fixed	ICE	tree-opt	ICE for skx in vect_build_gather_load_calls, at tree-vect-stmts.c:2835
40	102622	fixed	wrong code	tree-opt	Wrong code with -O1 and above due to phiopt and signed one bit integer types
41	102696	fixed	ICE	tree-opt	ICE in vect_build_slp_tree, at tree-vect-slp.c:1551 for skx and icx
42	102788	fixed	wrong code	tree-opt	Wrong code with -O3
43	102920	fixed	wrong code	tree-opt	Wrong code with -O3
44	103037	assigned	wrong code	tree-opt	Wrong code with -O2
45	103073	fixed	ICE	ipa	ICE in insert_access, at ipa-modref-tree.h:578
46	103122	fixed	ICE	tree-opt	ICE in fill_block_cache, at gimple-range-cache.cc:1277 with -O2
47	103361	fixed	ICE	tree-opt	ICE in adjust_unroll_factor, at gimple-loop-jam.c:407
48	103489	fixed	ICE	tree-opt	ICE with -O3 in operator[], at vec.h:889
49	103517	fixed	ICE	tree-opt	ICE in as_a, at is-a.h:242 with -O2 -march=skylake-avx512
50	103800	fixed	ICE	tree-opt	ICE in vectorizable_phi, at tree-vect-loop.c:7861 with -O3
51	104551	fixed	wrong code	tree-opt	Wrong code with -O3 for skylake-avx512, icelake-server, and sapphirerapids
52	105132	fixed	ICE	tree-opt	ICE in in operator[], at vec.h:889 with -march=skylake-avx512 -O3
53	105139	fixed	wrong code	target	GCC produces vmovw instruction with an incorrect argument for -O3
54	105142	fixed	wrong code	tree-opt	Wrong code with -O2
55	105189	fixed	wrong code	tree-opt	Wrong code with -O1
56	105587	fixed	ICE	target	ICE in extract_insn, at recog.cc:2791 (error: unrecognizable insn)

Table 3.1 continued

#	ID	Status	Type	Component	Description
57	106070	fixed	wrong code	tree-opt	Wrong code with -O1
58	106292	fixed	wrong code	tree-opt	Wrong code with -O3
59	106630	fixed	ICE	tree-opt	ICE: Segmentation fault signal terminated program cc1plus with -O2
60	106687	fixed	wrong code	tree-opt	Wrong code with -O2
61	107404	fixed	wrong code	target	Wrong code with -O3
62	108166	fixed	wrong code	tree-opt	Wrong code with -O2
63	108365	fixed	wrong code	c++	Wrong code with -O0
64	108647	fixed	ICE	tree-opt	ICE in upper_bound, at value-range.h:950 with -O3
65	109341	assigned	ICE	ipa	ICE in merge, at ipa-modref-tree.cc:176
66	109342	fixed	wrong code	tree-opt	Wrong code with -O2

Table 3.2: Reported LLVM bugs. YARPGen v.2 detected 28 bugs in LLVM. “ICE” stands for “Internal Compiler Error.” “RES” stands for “Resolved.” “CONFR” stands for “Confirmed.” We shortened names of the components and bug descriptions.

#	ID	Status	Type	Component	Description
1	42819	fixed	ICE	Backend: X86	ICE: "Cannot select: X86ISD::SUBV_BROADCAST" with -O3 -march=skx
2	42833	fixed	wrong-code	Backend: X86	Incorrect result with -O3 -march=skx
3	46178	fixed	ICE	Backend: X86	Assertion ‘idx <size()’ failed in combineX86ShufflesRecursively with -O3
4	46471	new	ICE	New Bugs	Assertion "Uses remain when a value is destroyed!"’ failed
5	46525	fixed	ICE	New Bugs	ICE: Assertion ‘!verifyFunction(*L->getHeader()->getParent())’ failed
6	46561	fixed	wrong-code	New Bugs	Wrong code with -O1 (Combine redundant instructions on function)
7	46586	fixed	wrong-code	Backend: X86	[x86] Clang produces wrong code with -O1
8	46661	fixed	ICE	New Bugs	ICE in backend: Instruction Combining stuck in an infinite loop after 100 iterations.
9	46680	fixed	ICE	ScalarOpt	ICE in backend: Instruction Combining stuck in an infinite loop after 100 iterations
10	46950	fixed	ICE	New Bugs	ICE: UNREACHABLE executed at llvm/lib/Transforms/Vectorize/LoopVectorize.cpp
11	47098	fixed	ICE	Opt	Polly crashes while running pass ’Polly - Forward operand tree’ on skx
12	47292	res	ICE	Opt	ICE in polly with -O3
13	48326	fixed	ICE	Backend: X86	Assertion "Invalid child # of SDNode!"’ failed with -march=skylake-avx512 -O3
14	48422	fixed	ICE	Opt	Assertion "Unknown counts for exiting blocks that dominate " "latch!"’ failed
15	48445	fixed	ICE	Opt	Assertion "Partial READ accesses not supported"’ failed
16	48554	fixed	ICE	isl	ICE: polly/lib/External/isl/isl_ast_build_expr.c:1745: cannot handle void expression
17	50109	fixed	ICE	Opt	UNREACHABLE executed at llvm/polly/lib/Transform/ManualOptimizer.cpp!
18	51797	new	ICE	New Bugs	ICE in backend: Instruction Combining stuck in an infinite loop after 100 iterations

Table 3.2 continued

#	ID	Status	Type	Component	Description
19	51798	fixed	ICE	LoopOpt	Assertion ‘hasVectorValue(Def, Instance.Part)’ failed
20	51906	new	wrong-code	LoopOpt	LICM introduces load in writeonly function (UB)
21	51923	new	ICE	LoopOpt	Clang segfaults with loop unroll(enable)
22	52002	new	ICE	New Bugs	Assertion "Uses remain when a value is destroyed!"’ failed
23	52273	new	ICE	LoopOpt	Assertion ‘!verifyFunction(*L->getHeader()->getParent(), &dbgs())’ failed.
24	52335	new	wrong-code	Backend: X86	Incorrect result with -O1 -march=skx
25	52504	fixed	ICE	Backend: X86	Assertion "Cannot use this version of ReplaceAllUsesWith!"
26	52560	fixed	ICE	Backend: X86	ICE in backend: Cannot select: t60: v8i16 = X86ISD::VZEXT_MOVL t55
27	52561	confr	ICE	Backend: X86	Assertion "Cannot BITCAST between types of different sizes!"’ failed
28	58616	fixed	ICE	New Bugs	Assertion "Expected vector-like insts only."’ failed

Table 3.3: Reported ISPC bugs. YARPGen v.2 detected 12 bugs in Intel® ISPC. “ICE” stands for “Internal Compiler Error.” We shortened names of the components and bug descriptions.

#	ID	Status	Type	Component	Description
1	1719	fixed	ICE	middle-end	Division by zero leads to ICE
2	1729	fixed	ICE	middle-end	Assertion failed: "ci != NULL".
3	1762	fixed	ICE	middle-end	ICE: "scatterFunc != NULL".
4	1763	fixed	wrong-code	middle-end	Wrong code for avx2-i64x4
5	1767	fixed	ICE	backend	Assertion ‘V.getNode() && "Getting TableId on SDValue()"’ failed.
6	1768	fixed	wrong-code	middle-end	Uniform and varying types have different rounding rules.
7	1771	fixed	wrong-code	backend	Wrong code for avx2-i64x4
8	1788	fixed	ICE	middle-end	LLVM ICE: Instruction Combining stuck in an infinite loop after 1000 iterations.
9	1793	fixed	wrong-code	backend	Wrong code for avx2-i32x16
10	1806	fixed	wrong-code	middle-end	ISPC produces wrong code with bool type iterator
11	1844	fixed	ICE	backend	LLVM ICE: "Unexpected illegal type" at llvm/lib/CodeGen/SelectionDAG/LegalizeDAG.cpp
12	1851	fixed	ICE	backend	LLVM assertion ‘Def == PreviousDef’ failed

same as that of depth two. As for array access patterns, we found that slicing along one of the array dimensions was the one that most commonly triggered bugs. We also found that the most buggy loop optimization component of GCC was related to vectorization, whereas in LLVM instruction selection was the buggiest component.

21% of the bugs we discovered were in a compiler's back end, despite the fact that our target is high-level loop optimizations. We believe that there are several factors involved here; for example, in some cases middle-end optimizations open up additional possibilities for back-end optimizations. Also, loop optimizations such as vectorization are tied closely to the compiler's back end, with details such as the presence of masking being important. A dedicated testing campaign for compiler back ends [45, 46] would almost certainly be more effective at finding back-end bugs than our approach is. However, that kind of approach risks finding less relevant bugs due to generating non-canonical IR, whereas our approach always presents back ends with canonical IR that is emitted by a compiler middle end.

3.4.1.2 Duplicate GCC bug reports. In several cases, we found a recently introduced GCC bug around the same time that others reported it. For example, bugs #103119, #105621, and #106605 were reported by compiler users. Additionally, another research group was running a similar compiler testing campaign concurrently with ours, but using mutation-based test-case generation. They found GCC bugs #103399, #106417, #107228, #108668 before we did. In all seven cases, we reported the bug within two days of others having reported it. Also, one bug (#96693) was erroneously marked as a duplicate despite the fact that we had reported it about a week earlier.

3.4.1.3 Impact of open-sourcing YARPGen v.2. A minor complicating factor in our testing campaign is that, during it, we released YARPGen v.2 as open-source software. We did this because we knew of several individuals who were specifically interested in fuzzing loop optimizations, and we judged that helping them (and others like them) out was more important than being the only users of our tool. For example, Martin Liška, a GCC developer, reported eight bugs that he found using our tool, including #101256. We did not know that they were using our tool, but we discovered this after the fact and included their

bugs in our list of GCC bugs. However, there may have been others using YARPGen v.2 whom we did not know about.

3.4.1.4 Speed of bug fixes in LLVM. Our testing campaign for LLVM did not go as well as the one for the GCC. Following standard practices for responsible external bug-finding, we avoided flooding their bug tracker with issues and instead kept the number of outstanding bug reports under a small limit. Alas, the LLVM developers often did not fix the issues that we reported very quickly, causing us to report fewer such bugs than we otherwise could have.

3.4.1.5 Partial bug fixes. An interesting facet of running a fuzzing campaign is observing cases where a fuzzer-triggered bug is fixed only incompletely. In a typical situation, the reported test case is fixed but without fully addressing the root cause of the bug; in this case, YARPGen v.2 finds other ways to trigger the issue soon after the initial, incomplete fix lands. This happened six times for GCC bugs during our testing campaign.

3.4.2 Covering Optimizations

A compiler fuzzer cannot find bugs in optimizations that it cannot trigger. Our loop generation policy mechanism is explicitly designed to trigger more loop optimizations more often; in this section, we evaluate its ability to do that. We use optimization counters[7] that the LLVM developers have provided in order to evaluate our ability to trigger optimizations. These counters are, in effect, a high-level, domain-specific code coverage metric.

3.4.2.1 Experimental setup. We threw out a number of counters that we judged to be irrelevant, including those for counting experimental optimizations, for bookkeeping unrelated to optimizations, and for counting optimizations that cannot be triggered from C or C++, such as those related to garbage collection. To do this we extracted all 1,360 counters from the LLVM source code and manually analyzed the resulting list in order to select those that we judged are related to loops, vectorization, or are enabled by other loop optimizations. Some of these counters were easy to identify (e.g., `loop-unroll.NumUnrolled` and `loop-vectorize.LoopsVectorized`); others required looking closely at the surrounding

[7] This is our term; LLVM simply calls them "statistics."

source code. In summary, we made a good-faith effort to pick a set of optimization counters that actually count loop-related optimizations. There are 238 of these.

We used LLVM 15.0.3[8] as the basis for these experiments. We compare YARPGen v.2's ability to trigger optimizations with that of a version of YARPGen v.2 where we disabled loop optimization policies, and also with LLVM's performance test suite.[9] This suite can optionally include various versions of the SPEC® CPU benchmark; we included SPEC® CPU2017 and configured it as directed by the LLVM developers.[10] For the benchmark suite, we simply compiled it and tallied up the number of times each optimization counter was triggered. For YARPGen v.2 (both with and without generation policies), we compiled randomly-generated programs over a 24-hour period using all cores on a machine with two AMD™ EPYC 7502 32-core processors.

3.4.2.2 Results of comparing YARPGen v.2 with the benchmark suite. Out of our set of 238 optimization counters, YARPGen v.2 (with generation policies) and the LLVM test suite, together, are able to trigger 80 of them. Eight were exclusively triggered by the test suite. Thus, YARPGen v.2 is able to test 90% of the optimizations that are triggered by the applications in LLVM's test suite (including SPEC® CPU2017).

3.4.2.3 Results of comparing YARPGen v.2 with and without generation policies. Only one of the 72 loop-related optimization counters (`gvn.MaxBBSpeculationCutoffReached-Times`) was exclusively triggered by YARPGen v.2 with generation policies; the rest were triggered by both versions. We used Welch's *t*-test to put each triggered optimization counter into one of three categories. For the first one, we can say with 95% confidence that generation policies are better (YARPGen v.2 with them triggers the counter more times than YARPGen v.2 without them); for the second category, we can say with 95% confidence that generation policies are worse; for the last one, the null hypothesis that neither version of YARPGen v.2 is better at triggering this particular counter. By this test, generation policies are better for 71 counters; worse for none; and, for one counter (`indvars.NumElimRem`) no

[8] https://github.com/llvm/llvm-project/releases/tag/llvmorg-15.0.3

[9] https://github.com/llvm/llvm-test-suite/releases/tag/llvmorg-15.0.3

[10] https://github.com/llvm/llvm-test-suite/tree/main/External/SPEC

conclusion can be drawn from our data. Table 3.4 shows a subset of these counters, along with the ratio between how many times each is triggered with and without generation policies. The geometric mean of this ratio across all 72 counters is 9.14. Overall, generation policies appear to be an effective way to trigger loop optimizations more often.

3.4.3 Code Coverage

Absolute code coverage numbers, for a compiler like GCC or LLVM, are tricky to interpret because these compilers support multiple source languages, multiple back ends, and many configuration options. On the other hand, in controlled circumstances, relative

Table 3.4: Optimization counters. Out of the 72 LLVM optimization counters that YARPGen v.2 can trigger, this table lists the most loop-relevant ones, and reports the average ratio between how many times that counter is triggered with generation policies (GP), as opposed to without them. For each of these counters, the data support the claim that generation policies trigger that counter more often, at a 95% confidence level.

Opt. counter name	GP to no GP ratio
licm.NumHoisted	10.29
licm.NumMovedCalls	3.48
licm.NumMovedLoads	20.31
licm.NumPromoted	12.34
licm.NumSunk	2.36
loop-delete.NumBackedgesBroken	2.04
loop-delete.NumDeleted	11.33
loop-idiom.NumMemSet	6.17
loop-instsimplify.NumSimplified	5.61
loop-peel.NumPeeled	2.40
loop-rotate.NumInstrsDuplicated	6.11
loop-rotate.NumInstrsHoisted	2.24
loop-rotate.NumNotRotatedDueToHeaderSize	4.69
loop-rotate.NumRotated	6.02
loop-simplify.NumNested	5.22
loop-simplifycfg.NumLoopBlocksDeleted	4.31
loop-simplifycfg.NumLoopExitsDeleted	9.29
loop-simplifycfg.NumTerminatorsFolded	6.90
loop-unroll.NumCompletelyUnrolled	10.97
loop-unroll.NumRuntimeUnrolled	9.79
loop-unroll.NumUnrolled	10.67
loop-unroll.NumUnrolledNotLatch	5.99
loop-vectorize.LoopsAnalyzed	4.09
loop-vectorize.LoopsEpilogueVectorized	11.55
loop-vectorize.LoopsVectorized	32.42

code coverage numbers might provide useful information. We collected code coverage for the LLVM 15.0.3[11] and GCC 12.2.0[12] implementations for the following inputs:

1. the unit test suite that is distributed with the compiler
2. SPEC® CPU2017 v1.0.1 (for GCC only)
3. the LLVM test suite, including SPEC® CPU2017 v1.0.1 (for LLVM only)
4. 24 hours of random testing with YARPGen v.2 in its default configuration, with the -O3 optimization flag, on an AMD™ Ryzen 9 5950X 16-core processor

The results are presented in Table 3.5 and Table 3.6. YARPGen v.2 does not improve the coverage by much, nor does it provide very good coverage by itself. However, these results are in line with previously reported code coverage due to generative random fuzzers (for example, Table 3 from the Csmith paper [6] and Tables 8 and 9 from the YARPGen v.1 paper [25]). Our view is that coverage of functions, lines, and branches are simply not very good metrics for evaluating compiler fuzzers—the internal behavior of compilers is highly path- and value-sensitive.

Table 3.5: Coverage of GCC source code.

	Functions	Lines	Branches
YARPGen v.2	37.28%	34.96%	23.79%
SPEC® CPU 2017	44.84%	40.96%	27.96%
unit tests	79.51%	77.09%	55.94%
unit tests + YARPGen v.2	79.86%	78.16%	57.40%
unit tests + SPEC®	79.63%	77.45%	56.36%
unit tests + SPEC® + YARPGen v.2	79.93%	78.34%	57.62%

Table 3.6: Coverage of LLVM source code.

	Functions	Lines	Branches
YARPGen v.2	22.30%	12.61%	12.37%
test suite (includes SPEC® CPU 2017)	27.99%	16.31%	17.14%
unit tests	84.26%	89.93%	73.58%
unit tests + YARPGen v.2	84.27%	89.97%	73.76%
unit tests + test suite	84.27%	89.97%	73.77%
unit tests + test suite + YARPGen v.2	84.28%	89.99%	73.85%

[11] https://github.com/llvm/llvm-project/releases/tag/llvmorg-15.0.3

[12] https://gcc.gnu.org/gcc-12/

3.4.4 Performance of YARPGen v.2

We measured the CPU time used by each step in the random testing pipeline, when testing LLVM 15.0.3 and GCC 12.2.0 at their `-O0` and `-O3` optimization levels. YARPGen v.2 was used in its default configuration. The CPU usage was measured from the scripting infrastructure that drives random testing, and it does not include the CPU time used by that infrastructure itself, but we do not believe it to be significant. We conducted this experiment on a machine with an AMD™ Ryzen 9 5950X 16-core processor, using all cores. The results are presented in Table 3.7. The majority of processor time in this experiment, 77.7%, was spent in the compilers, and only 0.78% was spent in YARPGen v.2. Thus, it is not a bottleneck during random testing.

3.5 Conclusion

YARPGen v.2 is an open-source[13] generative fuzzer that can be used to find bugs in loop optimizations in compilers for C, C++, ISPC, and SYCL. Over a three-year period, it was able to detect 122 wrong code bugs and internal compiler errors; most of these have been fixed. Moreover, we reported 13 GCC bugs that were independently rediscovered by users, showing that at least 20% of the bugs we reported for GCC are of the kind that users actually encounter and then (typically) triage and reduce in a painfully manual fashion.

The first contribution of this chapter is a novel static undefined behavior avoidance mechanism for loops that allows YARPGen v.2 to generate tests that are guaranteed to

Table 3.7: How CPU time is spent during random testing.

Tool	Step	% of total CPU time
YARPGen v.2	generation	0.78%
gcc -O0	compilation	12.38%
	execution	5.31%
gcc -O3	compilation	38.63%
	execution	5.27%
clang -O0	compilation	7.63%
	execution	5.47%
clang -O3	compilation	19.10%
	execution	5.43%

[13] https://github.com/intel/yarpgen

be compliant with language standards. Moreover, this method is applicable to multiple languages; combined with our syntax-independent IR, it allows us to target multiple languages with a single fuzzer. The second contribution is a collection of loop-specific generation policies that increase the number of applied loop and vector optimizations on average by 9.14 times for LLVM.

CHAPTER 4

PRELIMINARY RESEARCH ON UNIVERSAL COVERAGE-GUIDED COMPILER FUZZING VIA CHOICE SEQUENCE MUTATION

4.1 Introduction

Compilers play foundational roles in modern computing systems. Errors in them can propagate through the entire software stack, causing security vulnerabilities, data corruption, and even system crashes. Compiler crashes or hangs can be annoying, but when a compiler generates incorrect code, it can be dangerous. Therefore, a thorough testing of compilers is required. Compiler fuzzing has proved itself to be an effective testing technique for finding various bugs, including miscompilation errors.

Most of the existing compiler fuzzers are black-box fuzzers. They do not use any direct information about the software under test. Therefore, to increase testing effectiveness, black-box fuzzers usually employ various strategies to generate tests that are more likely to trigger bugs. Fuzzer developers have to utilize their knowledge about compilers and incorporate such knowledge into the fuzzer. Generation policies (Section 2.3.3 and Section 3.3.2) in YARPGen are a good example of this idea. Csmith used a set of carefully manually tuned probabilities that control test generation to generate tests that are more likely to stress-test the optimizing parts of the compiler.

Gray-box fuzzers, on the other hand, utilize some information about the software under test during the fuzzing process. They use this information in a feedback loop to guide the test generation process towards tests that are more likely to explore new parts of the compiler or trigger bugs. The most common type of feedback information is various kinds of code coverage. Coverage is used due to its simplicity and clear motivation. The main idea

behind it is that we cannot test parts of the software that we do not execute. Therefore, tests that achieve higher coverage are more desirable. General-purpose gray-box fuzzers received increased attention in the past decade after the introduction of AFL [19] and libFuzzer [47]. They were able to uncover a large number of bugs in various software. Currently, gray-box fuzzing is considered an industry standard for fuzzing, especially for security applications.

Compiler fuzzing, on the other hand, is still dominated by black-box fuzzers. There are very few implementations of gray-box compiler fuzzers, but each of them has its own limitations. We will discuss them in more detail in Section 6.6 and summarize the main downsides here. First, most of them operate either at the grammar level [48, 49] or the intermediate representation level [20, 21]. In both cases, they do not consider the full language standard. Therefore, they do not provide any guarantees about the generated tests, so produced tests do not always conform to the language standard. Such tests are not designed to detect miscompilation bugs or require third-party tools, such as sanitizers, to detect them. Second, each of the existing gray-box compiler fuzzers provides its own implementation. Thus, it is not easy to retrofit them to existing fuzzers. New coverage-guided compiler fuzzers have to be implemented from scratch.

Therefore, it would be desirable to create a coverage-guided fuzzing solution that can be easily adopted by multiple fuzzers and able to generate tests that comply with the language standard. This chapter investigates this research problem. We aim to design a universal coverage-guided compiler fuzzing framework that can be easily integrated into existing generative fuzzers with minimal effort. This will allow us to convert existing black-box compiler fuzzers into gray-box ones. We want to achieve this by replacing the source of randomness in the fuzzer with a *parametric generator* [50] that can be manipulated to guide the fuzzer.

This chapter of the dissertation describes the initial prototype. The prototype does not meet all the anticipated outcomes but provides valuable insights into encountered pitfalls. This information can be helpful to other researchers interested in exploring coverage-guided compiler fuzzing. We think it is important to share these negative results with the community to avoid the same mistakes in the future.

4.2 Background

Coverage-guided fuzzing, as the name suggests, uses code coverage to guide the fuzzing process. In essence, it works in the following way:

1. The software under test is instrumented to produce coverage information.
2. The fuzzer generates a test. It is usually done by mutating a test from the set of existing ones, called a corpus.
3. The test is executed on the software under test, and the coverage information is collected.
4. The fuzzer uses the coverage information to judge the effectiveness of the test. If the test uncovers new execution paths, it is added to the corpus for future mutations. Otherwise, it is discarded.
5. The fuzzer repeats steps 2-4 until the testing budget is exhausted.

The main advantage of coverage-guided fuzzing is that it guides the testing process toward the parts of the software under test that have not been explored yet. This allows us to achieve higher coverage with the same testing budget. The process is somewhat similar to stochastic gradient descent, where the gradient is replaced with the coverage information. Similarly to gradient descent, coverage-guided fuzzing relies on small steps to achieve the desired result. It means that mutations used to generate new tests should be small and localized. Otherwise, the fuzzer will not be able to utilize coverage information to guide the testing process effectively. The jumps between different parts of the test space will be too large, making it indistinguishable from pure random testing.

4.3 Implementation

4.3.1 Generative Fuzzing and a Source of Randomness

In principle, all the existing compiler fuzzers, except for the machine-learning-based ones (see Section 6.3), use the same approach to generate tests. They repeat the same step in a loop to create a new test until some end condition is met (e.g., test size). Every time multiple options are available during the generation process, the fuzzer makes a random choice. It uses a set of probabilities corresponding to each option and a source of randomness to make

the decision. Usually, the source of randomness is a pseudo-random number generator (PRNG) from the standard library. The PRNG is initialized with a seed, and it produces a sequence of pseudo-random numbers. The test generation process with a fuzzer that uses a PRNG is shown in Figure 4.1. Therefore, the whole test can be described by the sequence of random numbers that the fuzzer uses to make choices. There is a mapping between the sequence of random numbers and the test. We call such a sequence a *choice sequence*. It means that a change to the choice sequence will cause a change to the test. Thus, we can change the test by manipulating the choice sequence. To achieve this, we replace the default source of randomness with a parametric generator. The generator receives a choice sequence as a parameter and passes pieces of it to the fuzzer upon request.

Using a parametric generator for choice sequence manipulation provides many benefits and opens up new possibilities for random testing. First, it can be easily retrofitted to

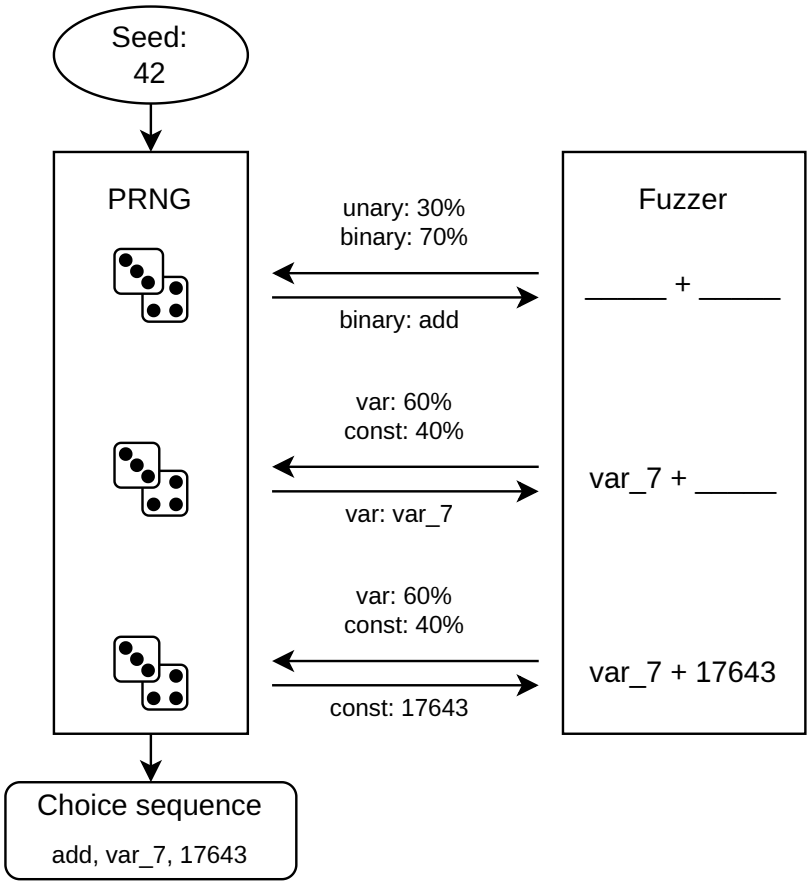

Figure 4.1: The test generation process with a fuzzer that uses a pseudo-random number generator (PRNG).

existing fuzzers. Many of them reached the saturation point and do not find new bugs anymore. For example, a recent paper that used Csmith as a baseline for comparison reported that it does not find new bugs in the latest versions of GCC and LLVM [18, 21]. Choice sequence manipulation allows us to apply new test generation strategies to existing fuzzers without rewriting them from scratch. Second, a parametric generator opens up a research opportunity to investigate new test space exploration strategies. For example, we can use a breadth-first search to prioritize generating diverse tests early. Another option is to use an exploration strategy that samples tests uniformly from the test space, avoiding any unintentional bias built into the fuzzer. One more option is to perform an exhaustive search of the test space, limiting the test size to a reasonable value. The motivation behind this idea comes from the observation that most of the fuzzing-reported bugs are relatively small after the reduction, so small tests can expose them [51]. These and other test space exploration strategies are currently under development by Regehr [52] and his colleagues. Third, choice sequence manipulation allows us to convert generative fuzzers into mutation-based ones. It allows us to combine the benefits of both approaches. Generative fuzzers are usually constructed in a way that guarantees that the generated tests are compliant with the language standard. The proposed conversion is achieved via mutations at the choice sequence level, so the resulting mutation-based fuzzer inherits the same guarantee. It means that we do not need to rely on third-party tools, such as sanitizers or formal interpreters, to check the compliance of the mutated tests. It allows us to perform a large-scale fuzzing campaign for software in domains where sanitizers and formal interpreters are unavailable, which is especially important for emerging languages. Moreover, independence from these tools allows us to use the internal reduction approach [53]. Conventionally, the reduction is performed by a separate tool, such as CReduce [29], that does not fully understand the language semantics. Therefore, it is not guaranteed that the reduced test complies with the language standard. Thus, conventional reducers have limited applicability without third-party tools. In contrast, internal reduction operates at the choice sequence level, so it utilizes the built-in guarantee of the generative fuzzer to ensure that all the reduced tests are compliant with the language standard. It allows us to perform automatic test reduction and makes it possible to perform a large-scale fuzzing campaign for software

in domains where sanitizers and formal interpreters are not available. Finally, we can use choice sequence manipulation to build a feedback loop between the fuzzer and the parametric generator. For example, we can follow the path of general-purpose fuzzers and use coverage information to guide the mutation of the choice sequence. It allows us to use the coverage information to explore the test space more efficiently. In theory, this approach should help us to revitalize the existing fuzzers that have reached the saturation point and push their limits even further. We chose to focus on coverage-guided fuzzing in this project because of its success in general-purpose fuzzing and its very clear motivation. The process of test mutation with a parametric generator and choice sequence manipulation is shown in Figure 4.2. Mutations play a crucial role in the coverage-guided fuzzing, and, as we discussed in Section 4.2, have strict requirements. Therefore, a substantial part of this project was dedicated to investigating the mutations of the choice sequence.

4.3.2 Choice Sequence Construction and Mutation

In general, mutations for coverage-guided fuzzing should be small and localized. They should be able to change a small part of the test without affecting the rest of it. Otherwise, the mutation process will be indistinguishable from a random generation of a new test. In this case, we will lose the ability to build tests that increase the coverage incrementally. Coverage-guided fuzzing with mutations that do not have such properties will be equivalent to black-box fuzzing, and we will not be able to utilize the coverage information in the testing process. Therefore, we need to find mutations of the choice sequence that satisfy these requirements. The main challenge in creating them comes from an additional level of indirection. The mutations have to operate at the choice sequence level, but the test is generated by the fuzzer. Small changes to the choice sequence can result in unpredictable changes to the test. Thus, this project's main challenge is figuring out the proper API between the fuzzer and the parametric generator. We want to be able to construct the choice sequence and mutations for it that can be used with different fuzzers. Additionally, the API should provide a correct level of granularity so the mutations on the choice sequence result in predictable changes to the test.

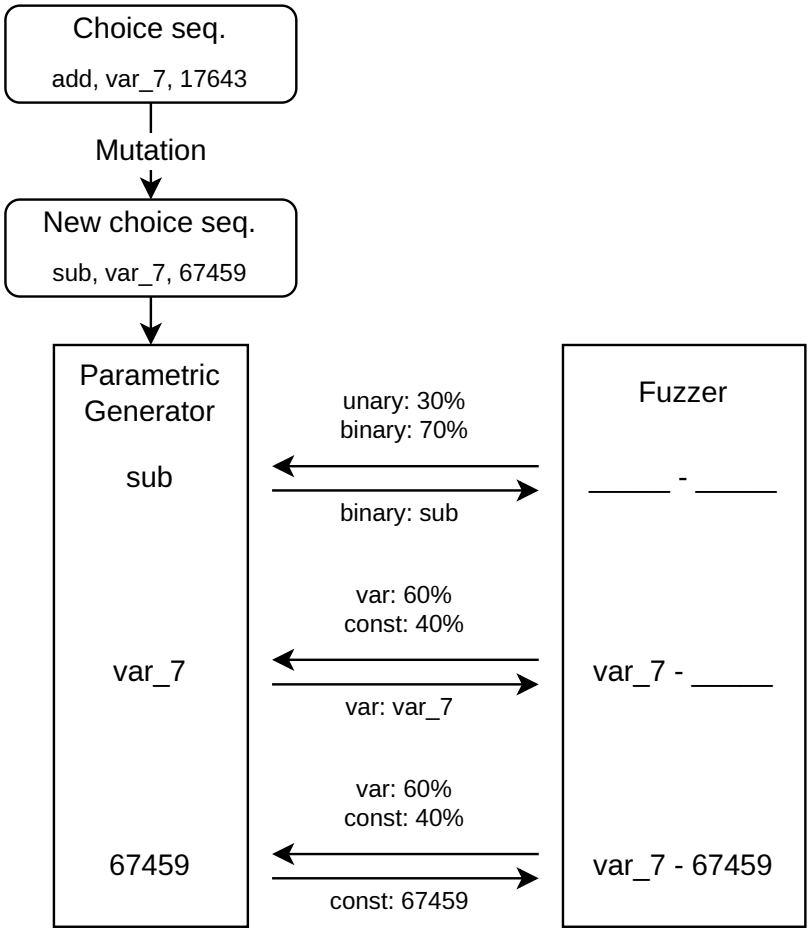

Figure 4.2: The process of test mutation with a parametric generator and choice sequence manipulation.

We developed a preliminary version of the API between the fuzzer and the parametric generator. It can certainly be improved, but we wanted to establish a baseline and check the feasibility of the approach. As of now, we have the following API:

- choose an element uniformly from a set of options;
- choose an element from a set of options with a given probability distribution;
- generate a random number from a given range.

The first two options are used to make decisions about significant structural elements of the test, such as types of expressions, kinds of arithmetic operations, or the number of statements in a block. The last option is used to generate general constants, such as integers in arithmetic expressions, and should be used for any decision that does not affect the structure of the test. This distinction is necessary to highlight the difference

between different building blocks of the test and is used during the mutation process. While some compiler optimizations depend on the presence of specific constants in the test (see Section 2.3.3.2), most of them, especially loop optimizations, are more affected by the structure of the test. Therefore, we want to prioritize the mutation of the choice sequence in a way that affects the structure of the test more than the constants.

Additionally, the fuzzer can inform the parametric generator when it starts and finishes generating some entity. For example, it can notify about the boundaries of the arithmetic expression or statement. It allows us to encode the hierarchy of random choices that the fuzzer makes during the generation process into the choice sequence, essentially creating a tree. This information is used during the mutation process.

Furthermore, we developed a set of mutations for the choice sequence that satisfy the requirements for coverage-guided fuzzing. They are relatively simple as of now because we want to establish a baseline for comparison and prove the concept. We implemented the following mutations:

- Replace a random number in the choice sequence with a new one. This mutation allows us to change the number of elements in the test (e.g., a number of statements in a loop body) or switch between different options from the same set (e.g., switch between different binary arithmetic operations).
- Delete a scope. This mutation can be used to remove a part of the choice sequence that could correspond to a loop or a conditional statement.
- Swap two scopes. It can be used to change the order of elements in the test (e.g., swap two statements in a block) without changing their content.
- Clone a scope. This mutation can be used to duplicate test elements (e.g., duplicate an arithmetic expression). The motivation for it comes from the common subexpression elimination optimization.

Additionally, we developed a resynchronization algorithm that helps us ensure that the mutations are localized. It uses information about the scope boundaries that the fuzzer provides to the parametric generator. We use it to realign the mutated choice sequence in a way that constrains the mutation to the scope where they occurred or nested scopes. It works in the following way. The parametric generator keeps track of the depth of the

current scope in the choice sequence and the depth of the current scope in the fuzzer during the generation process. As long as the two depths are equal, the generator returns the next element from the choice sequence. When the depth of the current scope in the choice sequence is greater than the one in the fuzzer, the generator creates new random choices until the two depths match. In the opposite case, the generator skips elements from the choice sequence until the two depths match. This approach is universal and works for any mutation. The developed parametric generator and mutations compose a prototype framework that can be used to perform coverage-guided fuzzing of compilers.

4.3.3 Coverage-Guided Framework

We decided to reuse an existing general-purpose coverage-guided fuzzing framework for compiler fuzzing. This way, we can avoid repeating the work required to run the software under test, collect coverage information, estimate the effectiveness of the test, and perform other bookkeeping tasks. We achieve this by integrating the parametric generator and mutations into the coverage-guided fuzzing framework by overriding the test mutation and execution steps. Let us explain the terminology we use in this section to avoid confusion. A choice sequence is a sequence of random numbers that the parametric generator uses to make decisions and communicate them to the test generator. A test is a code snippet that is generated by the test generator and passed to the compiler to collect coverage information. Fuzzer is a general-purpose coverage-guided fuzzing framework that we use to drive the testing process. A corpus is a set of choice sequences that the fuzzer uses to generate new tests. The whole setup works in the following way:

1. We initialize the corpus with a set of choice sequences that correspond to tests.
2. The fuzzer picks a choice sequence from the corpus and uses our custom mutations and resynchronization algorithm to generate a new test.
3. The mutated choice sequence is passed to the parametric and test generators to produce a new test.
4. The new test is compiled to collect coverage information and executed to check for miscompilation bugs.

5. The fuzzer uses the coverage information to estimate the effectiveness of the test and decides whether to add it to the corpus or discard it.
6. The fuzzer repeats steps 2-5 until the testing budget is exhausted.

Theoretically, the developed solution can use any general-purpose coverage-guided fuzzing framework, but we decided to use Centipede [54] for a couple of reasons. First, compiler fuzzing involves slow and large targets for which AFL and libFuzzer do not work well. Second, it allows the user to override the processing of the generated test easily. If we want to find miscompilation bugs, we need to compile the test and execute the resulting binary. We did not find a way to instruct the AFL to execute the resulting binary. This issue can be solved with proxy-fuzzing [21], but we prefer not to add another layer of complexity. Additionally, this ability simplifies the integration of the test generators that produce tests as multiple files (e.g., YARPGen).

Another important aspect of coverage-guided fuzzing is instrumentation of the software under test. We wanted to focus on the bugs in the optimizing parts of the compiler, so we decided to avoid fuzzing the front end. To achieve it, we used partial instrumentation and applied it only to the optimizing parts of the compiler.

The developed solution serves as a prototype for universal coverage-guided compiler fuzzing via choice sequence mutation. It allows us to investigate the effectiveness of the proposed approach and compare it to the existing compiler fuzzers.

4.4 Evaluation

This chapter describes the evaluation of the developed coverage-guided compiler fuzzing framework. As a test generator for the evaluation, we used YARPGen—a generative fuzzer for C-family languages. We chose it as a primary test generator because it is probably one of the most complex generative compiler fuzzers. We assumed that if YARPGen can be used with our framework, it should also be possible to use other generative compiler fuzzers.

4.4.1 Similarity of Mutated Programs

This experiment aims to evaluate if the proposed mutations and resynchronization techniques are effective in producing small and localized changes to the test. We compare the similarity between the original test and the mutated one to the similarity between the original test and a newly generated random test. The idea is that if the mutated test is more similar to the original one than the random test, then the mutations are effective.

We use `sim_c++` [55, 56] to measure the similarity between the original and mutated tests. We chose `sim_c++` because it is open-source and straightforward to use. It gives us a percentage of how many tokens of the mutated test are the same as in the original test. The higher the percentage, the more similar the tests are.

The experiment is performed in the following way. Initially, we generated a set of one million random tests using YARPGen. After that, we mutated each test in the set using the proposed mutations and collected the similarity measurement between the original and new tests. Additionally, for each test in the set, we generated a completely new random test and collected the similarity measurements between them. Finally, we tally up the results for each similarity measurement and plot them in Figure 4.3.

Of the mutated tests, 21% are identical to the original test, so they were not included in the plot. Out of the remaining, only 11% are entirely different from the original test, compared to 68% for the random tests. Additionally, 24% of the mutated tests are 90% similar to the original test. The results show that the proposed mutations have the potential to produce small and localized changes to the test, meaning that they can be used for coverage-guided fuzzing of compilers.

4.4.2 Coverage

This experiment aims to evaluate the effectiveness of the proposed coverage-guided compiler fuzzing framework and compare it to the existing random compiler fuzzers. As a target compiler, we chose LLVM because it is one of the most popular open-source compilers, and we are the most familiar with it. We use LLVM 17.0.6[1] for this experiment. We built it with `clang` using the following configuration options to collect coverage informa-

[1] https://github.com/llvm/llvm-project/releases/tag/llvmorg-17.0.6

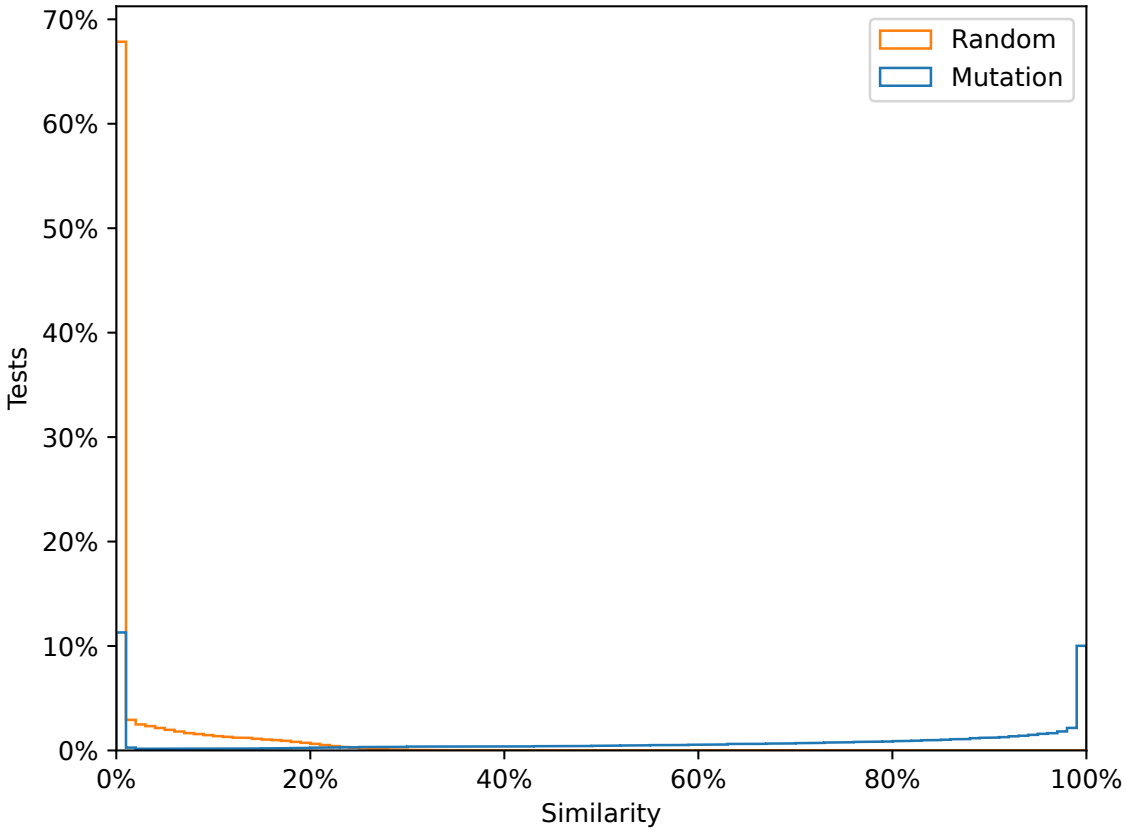

Figure 4.3: The similarity between the original test and the mutated one versus the similarity between the original test and a random test.

tion: `-fsanitize-coverage=trace-pc-guard,pc-table,trace-cmp`. Additionally, we used partial instrumentation to avoid fuzzing the compiler's front end. We instrumented only the optimization and analysis passes located under the `llvm` directory. The coverage is measured by combining the total number of covered edges,[2] comparison instructions, and switch statements.[3]

We normalized the results using the same testing budget for all the fuzzers. It means that the fuzzer gets exclusive access to the testing machine for a fixed amount of time. We used 120 hours for this experiment on a machine with two AMD™ EPYC 7502 32-core processors and 256 GB of RAM.

---

[2] https://releases.llvm.org/16.0.0/tools/clang/docs/SanitizerCoverage.html#tracing-pcs-with-guards

[3] https://releases.llvm.org/16.0.0/tools/clang/docs/SanitizerCoverage.html#tracing-data-flow

To collect the coverage information from the random compiler fuzzer, we have to execute tests produced by it under the coverage-guided fuzzing system. Unfortunately, it is not feasible to do it on a single machine because of the slowdown caused by the instrumentation and the overhead of the coverage-guided system. To overcome this issue, we used a cluster of 60 machines and the total number of tests to normalize the results. For random compiler fuzzing, we used YARPGen in its default configuration. First, we modified its testing system to only compile the test and not execute it. This was done to maximize the testing throughput and avoid the overhead of the test execution. The random compiler fuzzer's throughput averaged 19,200 tests per minute after two hours of testing. Next, we used this number to calculate the testing budget in terms of the number of tests. We chose to take a coverage snapshot every 12 hours of testing, which is equivalent to 13.8 million tests.

For the coverage-guided compiler fuzzer, we used YARPGen with the proposed coverage-guided fuzzing framework. We used the same testing budget as for the random compiler fuzzer and also took a coverage snapshot every 12 hours of testing. We started with a corpus that contained a single choice sequence. Additionally, we performed a separate experiment with a corpus containing distilled choice sequences from the random compiler fuzzer after 72 hours of testing and reduced the testing budget to match the original setup. The idea was to see if bootstrapping could help coverage-guided compiler fuzzers outperform random compiler fuzzers near the saturation point. The results are presented in Figure 4.4.

As we can see from the results, the coverage-guided compiler fuzzer has a noticeably worse performance than the random compiler fuzzer. The results mostly stay the same when we use the bootstrapped corpus. These results are contrary to our expectations and previous papers on coverage-guided compiler fuzzing (Section 6.6). There are several possible explanations for this. First, the instrumentation overhead might be too high for the coverage-guided compiler fuzzer to keep up with the random compiler fuzzer. In our case, the instrumented compiler is about 3x slower than the non-instrumented version. The coverage-guided fuzzing system causes additional overhead. It might be possible that random testing can explore the test space faster than the coverage-guided compiler fuzzer. Second, we might have chosen the timeout for compilation incorrectly. For this experiment, it was set to 5 minutes. It has been shown that programs with long execution times can

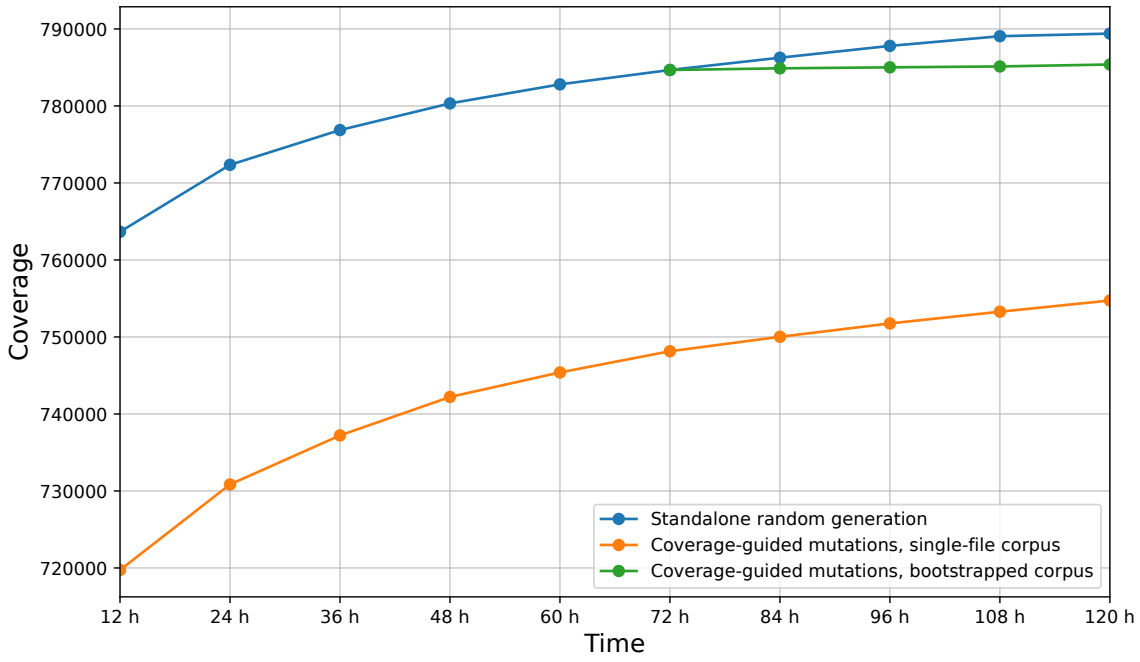

Figure 4.4: Comparison of the LLVM coverage achieved by the coverage-guided compiler fuzzer and the random compiler fuzzer. Coverage-guided mutations use an instrumented `clang` version, that is about 3x slower than the non-instrumented version.

consume up to 90% of the testing budget [57]. We noticed that the coverage-guided compiler fuzzer prefers to generate programs with long execution times, which might be the reason for the poor performance. The third option is that the proposed mutations are not effective enough to produce small and localized changes to the test. In this case, the test space exploration strategy will be similar to the random testing but with additional overhead. It means the coverage-guided compiler fuzzer will perform strictly worse than the random compiler fuzzer.

We acknowledge that the results presented in this section were derived from a single experiment and might not be representative of the general case. Ideally, we would like to perform more experiments to collect a statistically significant amount of data. However, the evaluation experiment takes a long time to run, and we were not able to do it within the time frame of this project. For example, to collect the data for the plot in Figure 4.4, we had to run the experiment for more than five days.

We do not have a definitive, clear explanation for these unexpected negative results. More thorough investigation and evaluation are required to understand the reasons behind

them. Additional experiments, such as a comparison of directed and undirected mutations, a comparison of random generation and coverage-guided fuzzing inside the coverage-guided compiler fuzzing framework, are needed to understand the reasons behind the unexpected results.

4.5 Conclusion

Overall, the idea of replacing the source of randomness in compiler fuzzers with a parametric generator looks promising. It opens up new possibilities for research and practical applications. For this chapter, we focused on the application of the idea to coverage-guided compiler fuzzing. We developed an initial prototype for the API between the parametric generator and the fuzzer and a set of mutations that can be used to guide the fuzzing process. We integrated our work into an existing general-purpose coverage-guided fuzzing framework. The project serves as a prototype for coverage-guided compiler fuzzing via choice sequence mutation. We evaluated the developed framework using YARPGen as a test generator. The results of the evaluation are mixed. On one hand, the proposed mutations are effective in producing small and localized changes to the test. On the other hand, the coverage-guided fuzzing framework cannot outperform the existing random compiler fuzzers. Further research is needed to understand the reasons behind the unexpected negative results that disagree with existing literature. We wanted to share these results with the community to help other researchers interested in exploring coverage-guided compiler fuzzing to avoid the same mistakes in the future.

CHAPTER 5

FUZZER DEVELOPMENT AND DEPLOYMENT INSIGHTS

The main purpose of this chapter is to summarize the findings of our multi-year experience of compiler fuzzing with YARPGen. It does not necessarily contain novel ideas but rather summarizes the lessons learned. We think that such information, collected in one place, can be useful for anyone who wants to embark on their own compiler fuzzing journey.

Multiple implementations of the same functionality are an excellent way to improve the fuzzer's effectiveness. Essentially, every time a fuzzer developer is not sure how to implement a particular feature, they should implement all possible options and select them at random. For example, YARPGen has multiple undefined behavior avoidance mechanisms for loops (Section 3.3.5) and chooses one to use randomly during the test generation. This approach allows the fuzzer to test all possibilities and find bugs that would be missed otherwise. This is one of the most important lessons we learned during our fuzzer development.

A test harness is an essential part of any fuzzer that strives to be useful in practice. Fuzzer developers often overlook this part of the random testing setup. A good testing harness takes the burden of constructing oracles, setting up the environment, and running the fuzzer off the user. It allows compiler developers to use the fuzzer out of the box. Moreover, it provides a consistent setup for the fuzzer, ensuring reproducibility and simplifying the debug process. Additionally, it serves as a cornerstone for further automation of the compiler random testing process. We think that usability played a significant role in the success of AFL.

Automation is a key to a continuous bug-finding process and, as a result, a large number of detected bugs. It allows the fuzzer to run continuously without any human intervention. In addition, automation allows developers to work on the fuzzer improvements instead of wasting time on tedious, repeated manual work. The testing system that comes with YARPGen (Section 2.3.4) is a good example of such automation. It is able to run the fuzzer, detect bugs, classify, deduplicate, and reduce them. As a result, we can launch the fuzzer and forget about it until it finds a bug. Moreover, we combined our testing system with a daily build of the compilers under test, allowing us to catch bugs early. We usually detect bugs within a week after they are introduced. Early bug detection is important because it allows compiler developers to address the issue while the memory of the change is still fresh. We would recommend starting automating the fuzzer as soon as possible. The steps required to detect and report a bug are often tedious and repetitive. The number of reported bugs can easily reach hundreds, so it is worth the effort.

Automatic test reduction systems are a crucial part of any large-scale fuzzing campaign. Without them, it would be impossible to create bug reproducers in a reasonable amount of time. For our bug-finding campaign, we primarily used C-Reduce [58] because it was the only available advanced reducer for C family languages at the time. However, that is no longer the case. Several new reducers were introduced over the past few years and are currently in active development. C-Vise [40] is a super-parallel C-Reduce port written in Python. Judging from our limited experience with it, C-Vise has a better performance but does not achieve the same reduction ratio as C-Reduce. Therefore, we would recommend using C-Vise for the initial reduction and then switching to C-Reduce for the final reduction. Another reducer that is worth mentioning is Shrink Ray [59]. It is in a very early stage of development but already shows promising results. Thus, we would recommend keeping an eye on it and trying it out when it becomes more mature.

Timeouts for various stages of the fuzzing process (test generation, compilation, and execution) can have a significant impact on the overall throughput of the fuzzer. The main idea behind the random testing is to run as many tests as possible, so a higher throughput is always better. A recent research paper [57] showed that programs with long execution times can consume over 90% of the total testing resources. Thus, it is important to select a

reasonable timeout for each stage or use a test selection mechanism proposed in the paper. Another way to improve the throughput is to avoid repeated work. For example, when YARPGen creates tests, it produces two separate files that require compilation: test function and test driver (Section 2.3.2.6). The test driver performs auxiliary work, such as variable initialization and checksum calculation. It is the same for all tests, so recompiling it for each compiler under test is wasteful. Thus, a good idea is to compile it once and then reuse the binary for all compilers.

Open-sourcing the fuzzer potentially reduces the number of bugs that the fuzzer can detect. It allows the compiler developers to use the fuzzer and fix the bugs during the development process. Therefore, fewer bugs make it to the trunk, and, as a result, fewer bugs are found by the fuzzer during external testing. However, open-sourcing the fuzzer also has many benefits. It enables collaboration with other researchers and developers and allows to build a community around the fuzzer. It is also a good way to get feedback and improve the fuzzer. To counteract the potential reduction in the number of reported bugs, we recommend reminding the fuzzer users to share their results as often as possible.

External fuzzing of open-source projects involves a surprisingly large amount of interaction with the developers. Our experience comes from compiler fuzzing, but we think that our observations apply to other domains as well. Overall, the end goal of any fuzzing campaign is to improve the quality of the software, and the developers are the ones who will determine the severity of the discovered bugs and whether they are relevant or not. Therefore, fuzzer developers should maintain a good relationship with the developers of the software under test. External fuzzing camping is a cooperative effort and can be successful only if both sides are willing to work together. To achieve it, we would highly recommend following responsible fuzzing guidelines [60]. The main takeaway from our experience is that it is important to ensure that submitted bugs are relevant and well-documented. We would also recommend being patient and keeping the number of submitted unaddressed bugs to a reasonable minimum. Otherwise, the bug tracking system will be flooded with bugs detected by fuzzer, and the developers could start ignoring them.

CHAPTER 6

RELATED WORK

Adapted from V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Art. no. 196, Nov. 2020. DOI: 10.1145/3428264 and V. Livinskii, D. Babokin, and J. Regehr, "Fuzzing loop optimizations in compilers for C++ and data-parallel languages," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Art. no. 181, Jun. 2023. DOI: 10.1145/3591295.

Our work on compiler testing is related to a large body of previous work. Therefore, we will direct the reader to the relevant overview papers for the in-depth review of compiler testing techniques in general and compiler fuzzing, in particular, and focus on the previous work most closely related to YARPGen in this chapter. Overview papers about various aspects of compiler testing and fuzzing were published by Chen *et al.* [4], Boujarwah and Saleh [61], Chen *et al.* [62], and Ma [63]. In this chapter, we focus on prior work that potentially finds the same kinds of errors as YARPGen or implements mechanisms similar to generation policies (Section 2.3.3 and Section 3.3.2). We will explicitly mention techniques that it uses to avoid undefined behavior in generated tests and work that is able to test loop optimizations.

6.1 Generative Compiler Fuzzing

To the best of our knowledge, a family of fuzzers called DDT [28], was the first one to target C compilers. In addition to that, the paper also introduced the term "differential testing." DDT was extremely expressive—it could generate all valid C programs—but at the expense of including undefined behaviors. The work had a limited mechanism for avoiding some kinds of undefined behavior during test-case reduction. However, it would not be suitable for finding miscompilation errors in modern compilers: the undefined operations

would raise too many false positives. Such false positives could be detected by sanitizers but would still overwhelm the testing system and use the majority of the testing time.

Quest [5] was specifically designed to test calling conventions and generated a great variety of interesting function call signatures. It avoided undefined behavior by simply not generating potentially dangerous constructs such as arithmetic expressions. Quest did not use differential testing but rather generated self-checking programs using assertions. YARPGen uses this self-checking approach as one of its oracles (Section 3.3.1). Overall, Quest and our fuzzer are almost completely disjoint in terms of bug-finding power.

Eide and Regehr improved the existing Randprog tool [26]. Their version used differential testing to look for miscompilation of volatile-qualified variables. Their testing method relied on the invariant that the number of loads from or stores to a memory location corresponding to a volatile-qualified variable in C should not change across compilers or optimization levels.

The Csmith [6] program generator avoided some undefined behaviors using static analysis but used wrapper functions that were evaluated at runtime to prevent unsafe arithmetic operations. In particular, dynamic runtime checks were used to eliminate UB in arithmetic operations and array subscripts. Subsequent research [10, 11] showed that wrapper functions impose a noticeable penalty on the code coverage and bug-finding ability of the fuzzer. This was one of the main motivations for YARPGen's preference for static UB avoidance. However, Csmith was more expressive than YARPGen in certain respects; it supported reasonably expressive pointer arithmetic, for example.

Ldrgen [64, 65] was designed to search for missed optimization opportunities. A motivating observation for this work comes from the fact that test cases generated by Csmith often contain a lot of dead code that is eliminated by the compiler, leading to a relatively small amount of machine code being emitted at the end. Ldrgen creates tests that are guaranteed to contain only live code using the Frama-C [66] static analyzer as a starting point. This generator does not avoid undefined behavior (except for limited attempts to avoid division by zero and oversized shifts). However, the generator has good expressiveness and it generates code that is guaranteed not to be eliminated by the compiler.

CLsmith [8] is a modified Csmith version created to test OpenCL compilers. Its test cases include vector types, intra-group communication, and atomic operations. Additionally, the developers supported equivalence modulo inputs [22]—also known as metamorphic testing—to further enhance the bug-finding ability of CLsmith.

The Orange family of fuzzers pioneered the static UB avoidance approach, developing this idea for generating branch-free scalar code using concrete value tracking over a series of papers [12–14, 67]. The first Orange test case generator simply discarded expressions that contained undefined behavior [12]; the second version added constants to shift expressions away from being undefined [13]; and the third one added *operation flipping* [14], a scheme for replacing operations that lead to undefined behavior with a related operation that avoids the problem. YARPGen builds directly upon this previous work but supports many additional language features (control flow, structures, arrays, pointers, etc.). The authors of Orange subsequently developed two ways to extend their UB avoidance mechanism to loops. The first one [68], effectively unrolls the loop inside the generator, analyzing all of its iterations. Any subexpression that causes UB at some iteration is augmented with array addition to avoid UB. For example, if `a[i] + b[i]` causes signed overflow for some value of `i`, it is replaced with `a[i] + (b[i] + tmp[i])`, where `tmp[i]` is initialized with zeroes except when a different value is required to avoid UB. This approach scales poorly in the presence of nested loops. Orange's second loop extension [43] allowed each loop to execute at most one iteration. The code generated by this approach is logically loop-free, but this fact can be hidden from the compiler by making the loop induction variable sufficiently opaque. YARPGen's approach is similar to these but improves upon them in several ways. In particular, YARPGen can generate loops that are executed multiple times with minimal overhead, allowing us to test deeply nested loops. Orange tracked all of the values in the test during the generation, so it used direct comparisons with the expected value as an oracle. This would make automated test case reduction difficult. YARPGen avoids this problem using differential testing as one of its oracles (Section 3.3.1).

GLSLsmith and WGSLsmith [69] were used to test OpenGL Shading Language (GLSL) and WebGPU Shading Language (WGSL) compilers, respectively. They introduced the idea of *program reconditioning*—a technique that eliminates undefined behavior from generated

programs after the generation process is complete. This approach is self-contained and does not require any external tools such as sanitizers. The main idea behind it is to replace operations that cause undefined behavior with code similar to Csmith [6] wrapper functions. Moreover, the authors of GLSLsmith and WGSLsmith used the program reconditioning technique to avoid floating-point roundoff errors. To the best of our knowledge, this is the only compiler fuzzer that was able to report a noticeable number of bugs related to floating-point arithmetic optimizations.

Generative fuzzers were used to test compilers for languages outside the C family. For example, TreeFuzz [70] is based on probabilistic, generative models. It uses a set of examples to capture properties of the input language. The authors used this fuzzer to test JavaScript and HTML. Glade [71] infers a target grammar from a set of examples and black-box access to a grammar parser. It was used to test Python, Ruby, and JavaScript. Dewey *et al.* [72] used Constraint Logic Programming to test the Rust typechecker. To increase expressiveness, they used an idea that is related to swarm testing [15]: instead of creating one generator that uses all of the language features, they created different generators for selected subsets of the language.

6.2 Mutation-Based Compiler Fuzzing

Mutation-based techniques have advantages and disadvantages with respect to the generation-based approach. The primary advantage of mutation is that since it builds on existing test cases, the expressiveness of the generated code is limited only by the expressiveness of the test cases that are used as the basis for mutation. The main disadvantage is that the generator is not stand-alone (for example, various EMI [22–24] efforts used programs generated by Csmith [6] as a starting point) and cannot easily guarantee the absence of undefined behavior in test cases. Our view is that the mutation-based and generation-based approaches to fuzzing are sufficiently different that they cannot be compared directly. Moreover, neither approach is likely to subsume the other: in practice, we need to employ both kinds of fuzzing.

Metamorphic testing [73] is the primary type of mutation-based approach that is used in compiler fuzzing. It is based on the idea that certain transformations, applied to a test

case, should not change the behavior of that test case. Segura *et al.* [74] provides a good overview of this technique and its applications. *Equivalence modulo inputs* (EMI) is a family of techniques that exploited this idea to find (as far as we know) more compiler bugs than any other single technique. Orion [22] pruned dead code from test cases, and Athena [23] had the capability to add dead code. Hermes [24] extended this technique by adding the ability to modify live code in a way that did not change the result of the test case.

Creal [75] is a recent mutation-based fuzzer that inserts real-world code fragments as mutations. It uses programs generated by Csmith [6] as seeds and ensures that inserted fragments are semantics-preserving. The authors of Creal collected a database of 51,356 code fragments from open-source projects that are used as mutations.

Metamorphic testing was also applied to test graphics shader compilers. GLFuzz [76] used it to test compilers for OpenGL shading language, GLSL. The authors of GLFuzz created a set of metamorphic relations that, similarly to YARPGen's generation policies, could be combined to increase the diversity of the generated tests.

6.3 Machine-Learning-Based Compiler Fuzzing

A recent trend in compiler fuzzing is incorporating machine learning (ML) into the generator. It has received increased attention recently, especially after the profound success of *large language models* (LLM) such as GPT-3 [77]. A summary paper about the application of machine learning techniques in fuzzing was published by Wang *et al.* [78]. ML-based fuzzing is not purely generative since it requires example programs for training purposes, but it does not fit cleanly into the mutation-based approach either.

For example, DeepSmith [79] uses the *long short-term memory* (LSTM) architecture of recurrent neural networks to target OpenCL compilers. The authors of DeepSmith trained their model on 10k OpenCL kernels from open-source repositories on GitHub. DeepSmith generates tests that contain undefined or non-deterministic behavior and then relies on third-party tools to filter them out. DeepFuzz [80] uses a sequence-to-sequence model trained on 10k tests sampled from the GCC test suite. It has a *pass rate* (the percentage of semantically valid programs that can be successfully compiled) of 82%. Montage [81] was the first ML-based fuzzer for finding vulnerabilities in JS engines. It uses the LSTM model

and achieves a pass rate of 80%. Dsmith [82] is able to generate C tests and was used to find bugs in GCC. It is based on LSTM architecture with an attention mechanism and was trained on 14k C programs from the compiler test suite. Dsmith achieves a maximal pass rate of 78%. ComFuzz [83] uses a pre-trained transformer-based model (DistilBERT [84]) that was fine-tuned on 10k programs extracted from open-source JS and Java projects hosted on GitHub. To increase the bug-finding ability of the fuzzer, ComFuzz uses historical test cases extracted from compiler tests and test suites as a starting point for the generation process. The motivation behind this approach is that historical test cases are more likely to trigger compiler bugs than purely randomly generated ones. In addition, the authors of ComFuzz equipped it with ten mutations to increase the diversity of the generated tests even further. Overall, ComFuzz achieves an average passing rate of 82%. WhiteFox [85] uses two LLMs for the test generation. The first one (GPT4 [86]) extracts optimization preconditions from the compiler source code. These preconditions are used to prompt the second LLM (StarCoder [87]) to generate tests. In addition to that, WhiteFox incorporates a feedback loop to enhance its testing ability further. Tests that successfully trigger an optimization are added to the prompt for the second LLM. WhiteFox was used to test PyTorch, TensorFlow Lite, TensorFlow-XLA, and LLVM. Similarly to WhiteFox, Fuzz4All [88] uses two LLLMs. It utilizes GPT4 to automatically summarize compiler documentation, code examples, specifications, or a combination of these and creates prompts for test generation. These prompts are passed to StarCoder to create actual tests. During the fuzzing process, Fuzz4All continuously updates the StarCoder's prompt to increase the diversity of the generated tests. Fuzz4All was used to test nine different systems for six different languages (C, C++, Go, SMT2, Java, and Python). It achieves a pass rate of 37% for C and 41% for C++. LLM-based fuzzer was also used to test Golang compilers [89]. It uses the pre-trained Codet5 [90] language model and a set of 1839 Go programs to fine-tune it. The resulting fuzzer was able to achieve a pass rate of 97%, and none of the generated programs displayed undefined behavior.

The machine-learning-based approach has a huge advantage: the structure of the generated programs is inferred rather than being painstakingly constructed by hand. However, it tends to produce programs that are not compliant with the relevant standards; for example,

UB-freedom is a global property that, so far, appears to be out of reach of learned generators. Of course, depending on the goals of a testing campaign, it might be desirable to present compilers with non-conforming inputs. Machine-learning-based test case generators claim to outperform conventional fuzzing techniques in terms of code coverage and bug-finding power, but these claims require further investigation.

The recent study by Nicolae *et al.* [91] performed an extensive evaluation of general-purpose ML-based fuzzers. The authors compared them with conventional fuzzers. They show that conventional grey-box fuzzers achieve higher code coverage, find more rare edges, and detect more bugs. Therefore, the original claims of ML-based fuzzers do not hold. Tian *et al.* [18] performed a similar evaluation for the compiler fuzzing domain. The authors of the paper argue that it is unfair to compare ML-based fuzzers with conventional ones. Conventional fuzzers are designed to produce well-defined test cases, and ML-based fuzzers usually do not have such restrictions. This imbalance inevitably threatens the fairness of the comparison. Thus, the authors of the paper propose a new baseline for evaluating ML-based compiler fuzzers called Kitten. They compare Kitten with two ML-based fuzzers, DeepSmith and DeepFuzz, and show that Kitten outperforms them in terms of code coverage, number of detected bugs, and the number of used language features. These contradictory results between independent evaluations of ML-based fuzzers and their original claims show that this topic requires further investigation. The nature of the compiler fuzzer comparison problem partially causes the disagreement. It is hard to compare different fuzzers fairly since they are designed to test different aspects of the compiler. Moreover, the evaluation might be necessary for academic purposes, but it is not always relevant for the practical application of the fuzzer in the industry. Compiler developers are usually interested in as many bugs as possible, and the best way to achieve this goal is to use as many different fuzzers as possible. Different fuzzers have different strengths and limitations, so they complement each other. Therefore, ML-based fuzzers give compiler developers an additional tool in their toolbox, and they should be used in conjunction with conventional fuzzers.

6.4 Towards Diversity in Randomly Generated Test Cases

An early approach to increasing the probability of triggering compiler optimizations is due to Burgess and Saidi [92]. Their fuzzer, which produced FORTRAN tests, explicitly introduced common subexpressions, linear induction variables, and arithmetic expression patterns that the optimizer was known to be looking for. Other optimizations, such as copy propagation, constant folding, algebraic optimizations, code motion, and dead code elimination, were assumed to happen due to the random nature of the generation process. Our generation policies build upon this idea. The paper used manually tuned probabilities in the generator in such a way that certain optimizations, such as constant folding and algebraic transformations, were triggered more frequently. YARPGen significantly extends this idea by supporting many more policies, including a large set of loop-specific generation policies, and by applying them automatically during the generation process (Section 2.3.3). The only shared feature between this previous work and YARPGen is the explicit generation of common subexpressions.

Swarm testing is an inexpensive way to improve test case diversity [15]. The idea is to randomly flip options in a generator that already exist to disable some of the language features in generated test cases. This way, one generated test might be completely free of bitwise operators, and another one might be dominated by them. In its original implementation, Swarm was designed as a stand-alone project that used the Csmith command-line interface to change the probability of generating some program element at the beginning of generating a test case. It could only alter global properties that were exposed by the developers externally. YARPGen has a similar built-in mechanism: parameter shuffling (Section 2.3.3.4). The motivation is the same: we want to increase diversity in generated tests by limiting or altering random choice decisions. Our implementation is significantly finer-grained since we built generation policies into YARPGen from the start. It helps us find hard-to-trigger bugs in optimizers.

History-guided configuration diversification (HiCOND) [16] changes the fuzzer configuration externally to increase the diversity of the generated tests, similar to swarm testing.

First, it uses historical testing data to infer the range for each parameter within the fuzzer that is likely to trigger a bug. Then, HiCOND uses a particle swarm optimization and novel diversity metric to find the set of optimal configurations that correspond to more diverse test cases. Finally, these configurations are used to guide the fuzzer.

Memoized configuration search (MCS) [93] is a machine-learning-based approach used to improve the diversity of generated tests. Similarly to swarm testing and HiCOND, it increases the diversity of generated tests by changing the fuzzer configuration. MCS uses multi-agent reinforcement learning to efficiently explore the space of possible configurations and shift the testing towards the ones that are more likely to trigger bugs. MCS was deployed by Huawei to test their proprietary C compiler.

Adaptive random testing [94, 95] is a technique for explicitly generating diverse test cases. The idea is that, to maximize test diversity, each new test case should be as dissimilar as possible to those that were already generated. This requires an explicit distance metric to be defined so that each new test can be chosen via distance maximization [96]. It was not clear to us whether there exists a sensible distance metric for the high-dimensional space occupied by C and C++ programs, and we did not pursue this direction in YARPGen.

6.5 Parametrized Generators

Zest [50] pioneered the idea of parametrized generators and applied it to QuickCheck-like random input generators to stress-test semantic analysis part of various programs. It also used a parametric generator to implement coverage-guided fuzzing. Zest inspired the creation of Clotho [97]— a Racket library for parametric randomness. It was designed to be flexible and easy to use. It provides a rich API to support various parametric generation strategies. Xsmith [98] is a fuzzer construction framework that allows users to create new differential fuzzers in a declarative way. It is implemented as a Racket library and domain-specific language and uses a parametric generator internally.

6.6 Coverage-Guided Compiler Fuzzing

Early implementations of coverage-guided compiler fuzzing for C++ operated at the grammar level. One of them was based on the libprotobuf-mutator library [48]. The other

one was prog-fuzz [49], an AFL-based fuzzer. Both of them were not able to produce tests that were compliant with the C++ standard, so they were able to find only a limited number of bugs. To overcome this issue, Fuzzil [20]—a coverage-guided fuzzer for JavaScript engines— developed a custom intermediate representation that is suitable for mutations. The IR was designed to simplify the mutations of control and data flow, allowing the fuzzer to produce semantically valid programs with a high probability.

GrayC [21]—a coverage-guided fuzzer for C compilers— developed a set of semantics-aware mutations that preserve the validity of generated tests with a high probability. These custom mutations allow GrayC to produce semantically valid tests in 99% of the cases. To the best of our knowledge, GrayC is the most advanced coverage-guided fuzzer for C compilers.

CHAPTER 7

CONCLUSION

This dissertation describes several novel techniques for generative compiler fuzzing. They are implemented in YARPGen to show their practical applicability. We think these techniques are not specific to our generator and can be applied to other fuzzers.

The first contribution of this dissertation is a novel static undefined behavior avoidance technique for C-family languages. It allows us to generate programs that are free of undefined behavior without any runtime checks, improving the bug-finding effectiveness of the fuzzer.

The second contribution is a collection of mechanisms called generation policies that allow us to target optimization explicitly, drastically increasing the bug-finding ability of the fuzzer. Our evaluation shows that this technique helps us find rare bugs.

The third contribution is a novel idea of multi-language support. It allows us to test multiple compilers with a single fuzzer. This technique is especially useful for testing compilers for emerging languages, where the amount of engineering budget required to build a fuzzer can be limited.

I implemented these techniques in our open-source fuzzer called YARPGen and performed an extensive multi-year fuzzing campaign of industry-grade compilers with it. As a result, I reported hundreds of bugs in GCC, LLVM, DPC++, ISPC, SDE, Alive2, and other projects. The majority of these bugs were fixed, indicating that bugs that YARPGen finds are deemed important by the compiler developers. YARPGen was also used independently of us by GCC and CompCert [99] developers, and by Intel, Apple, and HighTec EDV to test their compilers internally (we know this from private conversations). This shows that our fuzzer is recognized by the compiler community as a useful tool.

Moreover, YARPGen was used as a foundation for other research. For example, it was used as a basis for a tool that looks for missed optimizations [100]. Another research paper [57] used YARPGen along with Csmith as a compiler fuzzers examples to evaluate the impact of tests with long execution time on fuzzing throughput. In addition, YARPGen was used as a baseline for comparison in other research [85, 89]. All these examples show that YARPGen is recognized and used by the research community.

BIBLIOGRAPHY

[1] R. Isemann *et al.*, "Don't look UB: Exposing sanitizer-eliding compiler optimizations," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Art. no. 143, Jun. 2023. DOI: 10.1145/3591257.

[2] J. Lee *et al.*, "Taming undefined behavior in LLVM," *ACM SIGPLAN Not.*, vol. 52, no. 6, pp. 633–647, Jun. 2017. DOI: 10.1145/3140587.3062343.

[3] R. L. Sauder, "A general test data generator for COBOL," in *Proc. 1962 Spring Joint Comput. Conf. (AIEE-IRE)*, May 1–3, 1962, pp. 317–323. DOI: 10.1145/1460833.1460869.

[4] J. Chen *et al.*, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Art. no. 4, Feb. 2020. DOI: 10.1145/3363562.

[5] C. Lindig, "Random testing of C calling conventions," in *Proc. 6th Int. Symp. Autom. Anal. Driven Debugging (AADEBUG)*, Sep. 19–21, 2005, pp. 3–12. DOI: 10.1145/1085130.1085132.

[6] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, Jun. 4–8, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[7] B. Jiang *et al.*, "CUDAsmith: A fuzzer for CUDA compilers," in *2020 IEEE 44th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 13–17, 2020, pp. 861–871. DOI: 10.1109/COMPSAC48688.2020.0-156.

[8] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, Jun. 13–17, 2015, pp. 65–76. DOI: 10.1145/2737924.2737986.

[9] X. Yang, "Random testing of open source C compilers," Ph.D. dissertation, School Comput., Univ. Utah, Salt Lake City, UT, USA, 2015.

[10] K. Even-Mendoza, C. Cadar, and A. F. Donaldson, "Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour," in *2020 35th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 21–25, 2020, pp. 1219–1223. DOI: 10.1145/3324884.3418933.

[11] K. Even-Mendoza, C. Cadar, and A. F. Donaldson, "CSmithEdge: More effective compiler testing by handling undefined behaviour less conservatively," *Empir. Softw. Eng.*, vol. 27, no. 6, Art. no. 129, Jul. 2022. DOI: 10.1007/s10664-022-10146-1.

[12] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of C compilers targeting arithmetic optimization," in *Workshop Synth. Syst. Integr. Mixed Inf. Technol. (SASIMI)*, Mar. 8–9, 2012, pp. 48–53.

[13] E. Nagai, A. Hashimoto, and N. Ishiura, "Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers," in *Workshop Synth. Syst. Integr. Mixed Inf. Technol. (SASIMI)*, Oct. 21–22, 2013, pp. 88–93.

[14] E. Nagai, A. Hashimoto, and N. Ishiura, "Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 7, pp. 91–100, Aug. 2014. DOI: 10.2197/ipsjtsldm.7.91.

[15] A. Groce *et al.*, "Swarm testing," in *Proc. 2012 Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 15–20, 2012, pp. 78–88. DOI: 10.1145/2338965.2336763.

[16] J. Chen *et al.*, "History-guided configuration diversification for compiler test-program generation," in *2019 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 11–15, 2019, pp. 305–316. DOI: 10.1109/ASE.2019.00037.

[17] D. Amalfitano *et al.*, "Exploiting the saturation effect in automatic random testing of Android applications," in *2015 2nd ACM Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, May 16–17, 2015, pp. 33–43. DOI: 10.1109/MobileSoft.2015.11.

[18] Y. Tian *et al.*, "Revisiting the evaluation of deep learning-based compiler testing," in *Proc. 32nd Int. Joint Conf. Artif. Intell. (IJCAI)*, Aug. 19–25, 2023, pp. 4873–4882. DOI: 10.24963/ijcai.2023/542.

[19] Google, AFL, 2019. Accessed: Jan. 17, 2024. [Online]. Available: https://github.com/google/AFL.

[20] S. Groß, "FuzzIL: Coverage guided fuzzing for JavaScript engines," M.S. thesis, Karlsruhe Inst. Technol., Karlsruhe, Germany, 2018. [Online]. Available: https://saelo.github.io/papers/thesis.pdf.

[21] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "GrayC: Greybox fuzzing of compilers and analysers for C," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 17–21, 2023, pp. 1219–1231. DOI: 10.1145/3597926.3598130.

[22] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Not.*, vol. 49, no. 6, pp. 216–226, Jun. 2014. DOI: 10.1145/2666356.2594334.

[23] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Not.*, vol. 50, no. 10, pp. 386–399, Oct. 2015. DOI: 10.1145/2858965.2814319.

[24] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proc. 2016 ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, Nov. 2–4, 2016, pp. 849–863. DOI: 10.1145/2983990.2984038.

[25] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Art. no. 196, Nov. 2020. DOI: 10.1145/3428264.

[26] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proc. 8th ACM Int. Conf. Embed. Softw. (EMSOFT)*, Oct. 19–24, 2008, pp. 255–264. DOI: 10.1145/1450058.1450093.

[27] G. Ofenbeck, T. Rompf, and M. Püschel, "RandIR: Differential testing for embedded compilers," in *Proc. 2016 7th ACM SIGPLAN Symp. Scala*, Oct. 30–31, 2016, pp. 21–30. DOI: 10.1145/2998392.2998397.

[28] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998.

[29] J. Regehr *et al.*, "Test-case reduction for C compiler bugs," *ACM SIGPLAN Not.*, vol. 47, no. 6, pp. 335–346, Jun. 2012. DOI: 10.1145/2345156.2254104.

[30] *ISO/IEC 14882:2011, standard for programming language C++*, C++ Standards Committee, 2011. Accessed Jan. 17, 2024. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[31] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Nov. 8–13, 2020, pp. 713–724. DOI: 10.1145/3368089.3409729.

[32] D. B. Whalley, "Automatic isolation of compiler errors," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1648–1659, Sep. 1994. DOI: 10.1145/186025.186103.

[33] V. Livinskii, D. Babokin, and J. Regehr, "Fuzzing loop optimizations in compilers for C++ and data-parallel languages," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Art. no. 181, Jun. 2023. DOI: 10.1145/3591295.

[34] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time," *Int. J. Parallel Program.*, vol. 21, pp. 313–347, Oct. 1992. DOI: 10.1007/BF01407835.

[35] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *Int. J. Parallel Program.*, vol. 21, pp. 389–420, Oct. 1992. DOI: 10.1007/BF01379404.

[36] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in GCC and LLVM," *J. Syst. Softw.*, vol. 174, Art. no. 110884, Apr. 2021. DOI: 10.1016/j.jss.2020.110884.

[37] M. Pharr and W. R. Mark, "ISPC: A SPMD compiler for high-performance CPU programming," in *2012 Innov. Parallel Comput. (InPar)*, May 13–14, 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

[38] *SYCL™ 1.2.1 Specification*, Khronos® SYCL™ Working Group, 2020. Accessed Jan. 17, 2024. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf.

[39] N. P. Lopes *et al.*, "Alive2: Bounded translation validation for LLVM," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implement. (PLDI)*, Jun. 20–25, 2021, pp. 65–79. DOI: 10.1145/3453483.3454030.

[40] M. Liška, C-Vise, 2020. Accessed Apr. 2, 2022. [Online]. Available: https://github.com/marxin/cvise.

[41] C. Sun *et al.*, "Perses: Syntax-guided program reduction," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, May 27–Jun. 3, 2018, pp. 361–371. DOI: 10.1145/3180155.3180236.

[42] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[43] K. Nakamura and N. Ishiura, "Random testing of C compilers based on test program generation by equivalence transformation," in *2016 IEEE Asia Pacific Conf. Circuits Syst. (APCCAS)*, Oct. 25–28, 2016, pp. 676–679. DOI: 10.1109/APCCAS.2016.7804063.

[44] H. Jiang *et al.*, "CTOS: Compiler testing for optimization sequences of LLVM," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2339–2358, Feb. 2022. DOI: 10.1109/TSE.2021.3058671.

[45] Y. Rong, "IRFuzzer: Improving IR fuzzing with more diversified input," presented at the 2022 LLVM Developers' Meeting, Nov. 8–9, 2022. Accessed: Mar. 16, 2023. [Online]. Available: https://llvm.org/devmtg/2022-11/slides/Lightning2-ImprovingIRFuzzingWithMoreDiversifiedInput.pdf.

[46] N. Boushehri, "Automated translation validation for an LLVM backend," presented at the 2022 LLVM Developers' Meeting, Nov. 8–9, 2022. Accessed: Mar. 16, 2023. [Online]. Available: https://llvm.org/devmtg/2022-11/slides/TechTalk18-AutomatedTranslationValidation.pdf.

[47] LLVM, libFuzzer, 2017. Accessed Jan. 17, 2024. [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[48] K. Serebryany, V. Buka, and M. Morehouse, "Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator," presented at the 2017 LLVM Developers' Meeting, Oct. 18–19, 2017. Accessed: Jan. 16, 2024. [Online]. Available: https://llvm.org/devmtg/2017-10/slides/Serebryany-Structure-aware%20fuzzing%20for%20Clang%20and%20LLVM%20with%20libprotobuf-mutator.pdf.

[49] V. Nossum, Program Fuzzer, 2018. Accessed Jan. 25, 2022. [Online]. Available: https://github.com/vegard/prog-fuzz.

[50] R. Padhye *et al.*, "Semantic fuzzing with Zest," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 15–19, 2019, pp. 329–340. DOI: 10.1145/3293882.3330576.

[51] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," *ACM SIGPLAN Not.*, vol. 52, no. 6, pp. 347–361, Jun. 2017. DOI: 10.1145/3140587.3062379.

[52] J. Regehr, Guided-Tree-Search, 2022. Accessed Jan. 16, 2024. [Online]. Available: https://github.com/regehr/guided-tree-search.

[53] D. R. MacIver and A. F. Donaldson, "Test-case reduction via test-case generation: Insights from the Hypothesis reducer," *Leibniz Int. Proc. Inform.*, vol. 166, Art. no. 13, Nov. 2020. DOI: 10.4230/LIPIcs.ECOOP.2020.13.

[54] Google, Centipede, 2022. Accessed Jan. 17, 2024. [Online]. Available: https://github.com/google/fuzztest/tree/main/centipede.

[55] D. Grune and M. Huntjens, "Detecting copied submissions in computer science workshops," unpublished, 1989. [Online]. Available: https:/ / dickgrune . com / Programs/similarity_tester/Paper.pdf.

[56] D. Grune, The Software and Text Similarity Tester SIM, 1989. Accessed Jan. 17, 2024. [Online]. Available: https://dickgrune.com/Programs/similarity_tester/.

[57] J. Wu, Y. Yang, and Y. Zhou, "Boosting compiler testing via eliminating test programs with long-execution-time," in *2023 IEEE Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, Mar. 21–24, 2023, pp. 593–603. DOI: 10.1109/SANER56733.2023.00061.

[58] J. Regehr *et al.*, Creduce, 2012. Accessed Jan. 8, 2024. [Online]. Available: https://github.com/csmith-project/creduce.

[59] D. R. MacIver, Shrinkray, 2023. Accessed Jan. 8, 2024. [Online]. Available: https://github.com/DRMacIver/shrinkray.

[60] J. Regehr, "Responsible and effective bugfinding," *Embedded in Academia*, Aug. 17, 2022. Accessed: Oct. 17, 2022. [Online]. Available: https://blog.regehr.org/archives/2037.

[61] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: A survey and assessment," *Inf. Softw. Technol.*, vol. 39, no. 9, pp. 617–625, 1997. DOI: 10.1016/S0950-5849(97)00017-7.

[62] C. Chen *et al.*, "A systematic review of fuzzing techniques," *Comput. Secur.*, vol. 75, pp. 118–137, Jun. 2018. DOI: 10.1016/j.cose.2018.02.002.

[63] H. Ma, "A survey of modern compiler fuzzing," 2023. arXiv: 2306.06884.

[64] G. Barany, "Liveness-driven random program generation," in *Logic Based Program Synth. Transform. (LOPSTR)*, Oct. 10–12, 2018, pp. 112–127. DOI: 10 . 1007 / 978-3-319-94460-9_7.

[65] G. Barany, "Finding missed compiler optimizations by differential testing," in *Proc. 27th Int. Conf. Compil. Constr. (CC)*, Feb. 24–25, 2018, pp. 82–92. DOI: 10 . 1145 / 3178372.3179521.

[66] F. Kirchner *et al.*, "Frama-C: A software analysis perspective," *Form. Asp. Comput.*, vol. 27, no. 3, pp. 573–609, May 2015. DOI: 10.1007/s00165-014-0326-7.

[67] A. Hashimoto and N. Ishiura, "Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 9, pp. 21–29, Feb. 2016. DOI: 10.2197/ipsjtsldm.9.21.

[68] K. Nakamura and N. Ishiura, "Introducing loop statements in random testing of C compilers based on expected value calculation," in *Workshop Synth. Syst. Integr. Mixed Inf. Technol. (SASIMI)*, Mar. 16–17, 2015, pp. 226–227.

[69] B. Lecoeur, H. Mohsin, and A. F. Donaldson, "Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Art. no. 180, Jun. 2023. DOI: 10.1145/3591294.

[70] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," TU Darmstadt, Dept. Comput.

Sci., Darmstadt, Germany, Tech. Rep. TUD-CS-2016-14664, Nov. 2016. Accessed: Oct. 1, 2023. [Online]. Available: https://software-lab.org/publications/TreeFuzz_TR_Nov2016.pdf.

[71] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *ACM SIGPLAN Not.*, vol. 52, no. 6, pp. 95–110, Jun. 2017. DOI: 10.1145/3140587.3062349.

[72] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust typechecker using CLP (T)," in *2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 9–13, 2015, pp. 482–493. DOI: 10.1109/ASE.2015.65.

[73] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Hong Kong Univ. Sci. Technol., Dept. Comput. Sci., Hong Kong, Tech. Rep. HKUST-CS98-01, 1998. Accessed: Nov. 20, 2024. [Online]. Available: https://www.cse.ust.hk/~scc/publ/CS98-01-metamorphictesting.pdf.

[74] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Feb. 2016. DOI: 10.1109/TSE.2016.2532875.

[75] S. Li, T. Theodoridis, and Z. Su, "Boosting compiler testing by injecting real-world code," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Art. no. 156, Jun. 2024. DOI: 10.1145/3656386.

[76] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Art. no. 93, Oct. 2017. DOI: 10.1145/3133917.

[77] T. B. Brown *et al.*, "Language models are few-shot learners," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, Dec. 6–12, 2020, pp. 1877–1901.

[78] Y. Wang *et al.*, "A systematic review of fuzzing based on machine learning techniques," *PLoS One*, vol. 15, no. 8, Art. no. e0237749, Aug. 2020. DOI: 10.1371/journal.pone.0237749.

[79] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 16–21, 2018, pp. 95–105. DOI: 10.1145/3213846.3213848.

[80] X. Liu, X. Li, R. Prajapati, and D. Wu, "DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing," *Proc. AAAI Conf. Artif. Intell.*, vol. 33, no. 1, pp. 1044–1051, Jul. 2019. DOI: 10.1609/aaai.v33i01.33011044.

[81] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided JavaScript engine fuzzer," in *Proc. 29th USENIX Conf. Secur. Symp.*, Aug. 12–14, 2020, Art. no. 147.

[82] H. Xu *et al.*, "DSmith: Compiler fuzzing through generative deep learning model with attention," in *2020 Int. Joint Conf. Neural Networks (IJCNN)*, Jul. 19–24, 2020, pp. 1–9. DOI: 10.1109/IJCNN48605.2020.9206911.

[83] G. Ye *et al.*, "A generative and mutational approach for synthesizing bug-exposing test cases to guide compiler fuzzing," in *Proc. 31st ACM Joint Meeting Eur. Softw.*

Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), Dec. 3–9, 2023, pp. 1127–1139. DOI: 10.1145/3611643.3616332.

[84] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2020. arXiv: 1910.01108.

[85] C. Yang *et al.*, "WhiteFox: White-box compiler fuzzing empowered by large language models," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Art. no. 296, Oct. 2024. DOI: 10.1145/3689736.

[86] J. Achiam *et al.*, "GPT-4 technical report.," 2023. arXiv: 2303.08774.

[87] R. Li *et al.*, "StarCoder: May the source be with you!" 2023. arXiv: 2305.06161.

[88] C. S. Xia *et al.*, "Fuzz4All: Universal fuzzing with large language models," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, May 27–Jun. 3, 2024, Art. no. 126. DOI: 10.1145/3597503.3639121.

[89] Q. Gu, "LLM-based code generation method for Golang compiler testing," in *Proc. 31st ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Dec. 3–9, 2023, pp. 2201–2203. DOI: 10.1145/3611643.3617850.

[90] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021. arXiv: 2109.00859.

[91] M.-I. Nicolae, M. Eisele, and A. Zeller, "Revisiting neural program smoothing for fuzzing," in *Proc. 31st ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Dec. 3–9, 2023, pp. 133–145. DOI: 10.1145/3611643.3616308.

[92] C. J. Burgess and M. Saidi, "The automatic generation of test cases for optimizing Fortran compilers," *Inf. Softw. Technol.*, vol. 38, no. 2, pp. 111–119, Sep. 1996. DOI: 10.1016/0950-5849(95)01055-6.

[93] J. Chen *et al.*, "Compiler test-program generation via memoized configuration search," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.*, May 14–20, 2023, pp. 2035–2047. DOI: 10.1109/ICSE48619.2023.00172.

[94] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The ART of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, Jan. 2010. DOI: 10.1016/j.jss.2009.02.022.

[95] R. Huang *et al.*, "A survey on adaptive random testing," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2052–2083, Sep. 2021. DOI: 10.1109/TSE.2019.2942921.

[96] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 26–30, 2019, pp. 142–153. DOI: 10.1145/3338906.3338970.

[97] P. Darragh, W. G. Hatch, and E. Eide, "Clotho: A Racket library for parametric randomness," in *Proc. 2020 Scheme Funct. Program. Workshop*, Aug. 28, 2020, pp. 3–13.

[98] W. Hatch *et al.*, "Generating conforming programs with Xsmith," in *Proc. 22nd ACM SIGPLAN Int. Conf. Gener. Program. Concepts Exp. (GPCE)*, Oct. 22–23, 2023, pp. 86–99. DOI: 10.1145/3624007.3624056.

[99] D. Monniaux, L. Gourdin, S. Boulmé, and O. Lebeltel, "Testing a formally verified compiler," in *Tests Proofs (TAP)*, Jul. 18–19, 2023, pp. 40–48. DOI: 10.1007/978-3-031-38828-6_3.

[100] Y. Zhang, "Detection of optimizations missed by the compiler," in *Proc. 31st ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Dec. 3–9, 2023, pp. 2192–2194. DOI: 10.1145/3611643.3617846.