A Survey of Compiler Testing

JUNJIE CHEN, College of Intelligence and Computing, Tianjin University, China
JIBESH PATRA and MICHAEL PRADEL, Department of Computer Science, University of Stuttgart, Germany
YINGFEI XIONG, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
HONGYU ZHANG, School of Electrical Engineering and Computing, University of Newcastle, Australia
DAN HAO and LU ZHANG, Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China

Virtually any software running on a computer has been processed by a compiler or a compiler-like tool. Because compilers are such a crucial piece of infrastructure for building software, their correctness is of paramount importance. To validate and increase the correctness of compilers, significant research efforts have been devoted to testing compilers. This survey article provides a comprehensive summary of the current state-of-the-art of research on compiler testing. The survey covers different aspects of the compiler testing problem, including how to construct test programs, what test oracles to use for determining whether a compiler behaves correctly, how to execute compiler tests efficiently, and how to help compiler developers take action on bugs discovered by compiler testing. Moreover, we survey work that empirically studies the strengths and weaknesses of current compiler testing research and practice. Based on the discussion of existing work, we outline several open challenges that remain to be addressed in future work.

CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**; *Maintaining software*;

Additional Key Words and Phrases: Compiler testing, test program generation, test oracle, test optimization, compiler debugging

ACM Reference format:
Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (February 2020), 36 pages.
https://doi.org/10.1145/3363562

This work was supported by the National Key Research and Development Program of China under Grant No.2017YFB1001803 and the National Natural Science Foundation of China under Grant Nos. 61672047, 61861130363, 61872008, and 61828201. This work was also supported by BMWF/Hessen within CRISP and by the German Research Foundation within the ConcSys and Perf4JS projects. Yingfei Xiong and Dan Hao are the corresponding authors. This work was mainly done when Junjie Chen was at Peking University. Yingfei Xiong, Dan Hao, and Lu Zhang are also affiliated with Department of Computer Science and Technology, Peking University, China.
Authors' addresses: J. Chen, College of Intelligence and Computing, Tianjin University, Tianjin, 300350, China; email: junjiechen@tju.edu.cn; J. Patra and M. Pradel, Department of Computer Science, University of Stuttgart, Stuttgart, 70569, Germany; emails: jibesh.patra@gmail.com, michael@binaervarianz.de; Y. Xiong, D. Hao, and L. Zhang, Department of Computer Science and Technology, Peking University, Beijing, 100871, China; emails: {xiongyf, haodan, zhanglucs}@pku.edu.cn; H. Zhang, School of Electrical Engineering and Computing, University of Newcastle, NSW, 2308, Australia; email: hongyu.zhang@newcastle.edu.au.

0360-0300/2020/02-ART4 $15.00
https://doi.org/10.1145/3363562

1 INTRODUCTION

Compilers are important tools, because they are a central piece of infrastructure for building other software. Virtually every program that runs on a computer, ranging from operating systems over web browsers to small scripts written by end-users, has been processed by a compiler or a compiler-like tool. Because compilers are such a central piece of infrastructure, they are very widely distributed. For example, popular compilers of widespread programming languages, such as GCC for C/C++, are run by millions of users. Beyond these direct users of compilers, typically developers, much more people indirectly rely on compilers when executing compiled programs.

Even though considerable efforts have been made to improve the quality of compilers, similar to all other software, compilers still contain bugs [72, 103, 123]. A compiler bug can cause incorrect binary code to be generated from correct source code. Also, a single bug in a production compiler can propagate to any application built upon it and cause surprising and possibly harmful misbehavior. For example, a miscompilation bug in the Java 7 implementation caused several popular Apache projects to crash.[1] Sometimes, compiler bugs may even be injected on purpose to compromise the security of compiled applications. For example, a malicious variant of Apple's Xcode development environment contained a compiler "bug" that introduces a backdoor into every compiled application.[2] Such compiler backdoors can also exploit accidentally introduced bugs, as evidenced by work on compromising the Unix *sudo* tool via a publicly known bug in LLVM [105].

Compiler bugs not only cause unintended behavior with possibly severe consequences, but also make software debugging more difficult. The reason is that developers can hardly determine whether a software failure is caused by the program they are developing or the compilers they are using [26]. For example, when a buggy compiler optimizes a correct program into an executable that has incorrect runtime behavior, it is unclear to the developer of the program what causes the unexpected behavior. Since application developers usually assume the misbehavior tends to be caused by bugs introduced by themselves, they may spend a long time to eventually realize that a compiler bug is the root cause.

Given the importance of compilers, there is a huge interest in implementing them correctly. However, reaching this goal is non-trivial, since compilers are complex pieces of software. A typical compiler consists of a pipeline of interacting components that address, e.g., the lexical analysis of the source language, parsing the source language into an intermediate representation, applying semantic checks, optimizing the code, and generating code in the target language. Currently, the optimization phase and implementation of parallelism and object-oriented features make it more complex. Because of this complexity, traditional computer science curricula typically dedicate at least one entire course to the art of constructing compilers. In contrast to other complex software, the domain that compilers deal with is particularly rich. Both the input and the output of compilers usually are programs written in Turing-complete languages. Moreover, both the inputs and output domains are infinitely large, since programs can be arbitrarily long. As a result, reasoning about the behavior of a compiler is all but trivial.

The difficulties of correctly implementing a compiler lead to several challenges for testing the implementation. One challenge is the lack of a formal specification of what exactly a compiler is supposed to do. While the high-level specification is implicitly known—compilers translate a source program into a target program in a semantics-preserving way—the lower level details are usually unspecified. For instance, when to apply what optimizations is rarely specified, making it difficult to check whether a compiler applies all desired optimizations. As an example, the LLVM

[1] http://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html.
[2] https://arstechnica.com/information-technology/2015/09/apple-scrambles-after-40-malicious-xcodeghost-apps-haunt-app-store/.

compiler has 58 optimizing transformation passes,[3] which can be combined in various ways, but no specification exists when to apply which of these passes.

Another challenge for testing is the semantic richness of the input and output languages that compilers deal with. Because source code can express a wide range of computations, including many computations that cause the program to crash early, automatically creating source code with non-trivial behavior is difficult. However, such non-trivial programs are often required to reach deep into the pipelined workflow of a compiler and to test hard-to-reach code. For example, an input program must pass all sanity checks and type checks in the lexer and parser before it can test optimizations performed by the compiler. However, a non-compiler program often does not have such complex checks. A related challenge is that small changes in the input can cause huge differences in the expected behavior. For example, changing a single character in an input program may cause the compiler to take a completely different path and to expose completely different behaviors.

A third important challenge is that compilers have various options and features. For example, most compilers offer different optimization levels, support several variants of the source language, and consider multiple target platforms. This multitude of configurations creates a large space that is difficult to explore exhaustively and orthogonal to the regular input space.

In addition to these challenges, compilers fortunately also have some properties that simplify the problem of validating their correctness. One such property is that the inputs to compilers are written in a programming language, i.e., the space of possible inputs is clearly defined by the language grammar. In contrast, general-purpose fuzzing tools, such as AFL,[4] may not have a grammar to help control the generation of syntactically valid inputs. Another property that eases compiler testing is that the semantics of the source language are usually specified, either informally in a language specification or formally, e.g., for a core of the programming language. As a result, the expected behavior of (most) programs and thereby also of the compiler output, is known when running the program. Finally, compiler testing benefits from the fact that for most popular programming languages, there are multiple supposedly equivalent implementations, which compiler testing can exploit as an oracle for differential testing [81] (discussed in detail in Section 4).

Motivated both by the importance of compilers and by the interesting challenges involved in improving their reliability and correctness, the area has received significant attention from researchers and practitioners. A particularly successful line of work is compiler testing, which has made tremendous advances in recent years. Several widely adopted tools and approaches have been developed in the past decade, some of which found hundreds of bugs in production compilers of popular languages [72, 123].

The impressive progress made in compiler testing is good news for developers and users of compilers. However, the huge amount of existing work makes it difficult for interested non-experts to understand the state-of-the-art and how to improve upon it. This article summarizes existing work and provides a retrospection of the compiler testing field after years of development. The readers can thus have an in-depth understanding of the advantages and limitations of the existing approaches. Our findings can help researchers and practitioners understand more about the implications of the existing compiler testing approaches and help prompt their adoption in practice. Based on our analysis, we also point out the current challenges and suggest possible future directions for compiler testing research.

We present the work on compiler testing from the following six perspectives, as illustrated in Figure 1:

[3] https://llvm.org/docs/Passes.html.
[4] http://lcamtuf.coredump.cx/afl/.

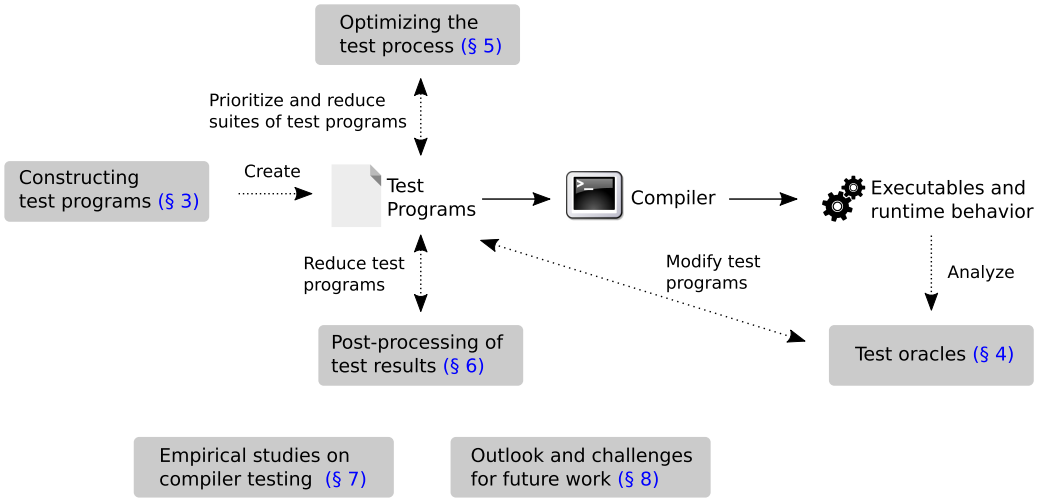

Fig. 1. Overview of topics covered in this article.

- Constructing test programs: Section 3 presents approaches to creating programs that serve as inputs for testing compilers.
- Test oracles: Section 4 describes approaches to determine whether the behavior or output of a compiler is correct or not.
- Optimizing the test process: Section 5 discusses how to make compiler testing more efficient.
- Post-processing of test results: Section 6 presents approaches that help developers prioritize and understand bugs detected through compiler testing.
- Empirical studies: Section 7 discusses empirical studies performed to better understand compiler bugs and the process of compiler testing.
- Outlook and challenges for future work: Section 8 presents open challenges that remain to be addressed by future work.

Orthogonal to the efforts on compiler testing discussed in this article, formally verifying compilers is another attempt towards more reliable compilers. Work on compilers verification [36] checks whether the implementation of a compiler complies with a formal specification of some correctness properties, such as, to not crash during compilation or to preserve the semantics of the input program in the produced machine code. A particularly noticeable example is the CompCert compiler [76], which targets a subset of the C language and which has been formally verified. For this article, we focus on compiler testing and do not cover work on compiler verification in detail.

2 SURVEY METHODOLOGY

For this survey, we carefully collect 85 papers from relevant international journals and conferences. To collect these papers, we systematically search the DBLP publication database[5] using the following keywords: "compiler test," "compiler validation," "compiler defect," "compiler bug," "compiler vulnerabilit(y/ies)," "compiler fault," "compiler error," "compiler issue," and "compiler debug." Then, we manually filter the results by removing irrelevant papers. Finally, we retrieve additional papers by following references in the already found papers.

[5] https://dblp.uni-trier.de.

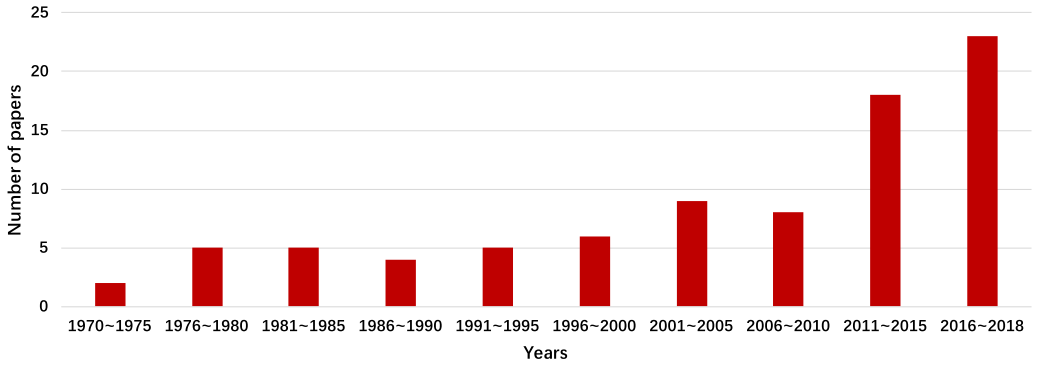

Fig. 2. Compiler testing papers from 1970 to 2018.

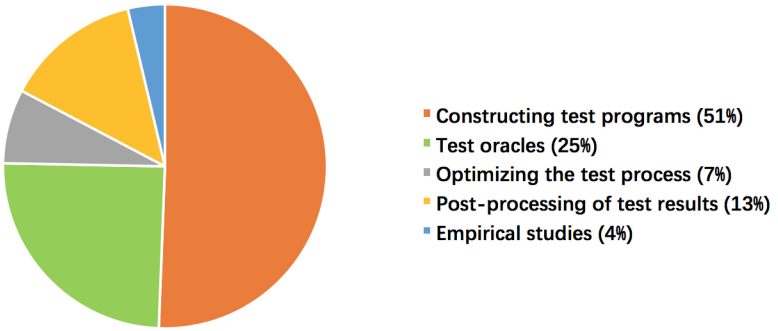

Fig. 3. Paper distribution on each research perspective.

Figure 2 shows the collected papers from 1970 to 2018. We can see that the number of relevant papers has increased significantly since 2011, indicating that the problem of compiler testing has received significant attention since then. We further analyze the reason behind this phenomenon. One possible reason is that some easy-to-use and effective tools such as Csmith [123] and C-Reduce [97] were developed and released around 2011, since nearly 50% papers from 2011 to 2018 are based on these tools. Actually, this is as expected. For example, constructing test programs is one of the most important challenges in compiler testing and is the initial step of the testing process. Csmith makes this step convenient, and thus promotes compiler testing research.

Figure 3 shows the distribution of papers on each research perspective, including constructing test programs, test oracles, optimizing the test process, post-processing of test results, and empirical studies. If a paper addresses more than one research perspective, then we categorize it based on its most important contribution. Figure 3 shows that, like general software testing, most papers on compiler testing also focus on constructing test programs and test oracles. In particular, constructing test programs attracts most of the attention. This is as expected, because it is the initial step of the testing process (only after test programs are constructed can the testing process be started).

Furthermore, we are aware of four existing surveys on compiler testing, all of which were published before 2005. Burgess [15] summarizes the main automatic compiler testing approaches before 1994. Boujarwah and Saleh [13] assess and compare various compiler test-program construction approaches before 1997. Tonndorf [117] presents tool-based Ada compiler validations before 1998. Kossatchev and Posypkin [68] present compiler testing approaches based on formal specifications of the programming language syntax and semantics before 2005. These existing survey

Table 1. Overview of Approaches for Constructing Test Programs

Constructing Test Programs (Section 3)		**Approach**
Manually Constructing Test Programs (Section 3.2)		Callahan et al. [17], Dongarra et al. [44], Wolf et al. [120]
Test Program Generation (Section 3.3)	Grammar-directed Approaches (Section 3.3.1)	Purdom [96], Hanford [52], Houssais [61], Duncan and Hutchison [45], Burgess [34], Burgess and Saidi [16], Bird and Munoz [12], Amodio et al. [4], Bazzichi and Spadafora [9], Zelenov et al. [127], Zelenov and Zelenova [126], Lindig [78, 79]
	Grammar-aided Approaches (Section 3.3.2)	Sirer et al. [108], Yang et al. [123], Morisset et al. [85], Lidbury et al. [77], Alipour et al. [3], Holler et al. [58], Boujarwah et al. [14]
	Other Approaches (Section 3.3.3)	Berry [11], Mandl [80], Eide and Regehr [47], Nagai et al. [87], Palka et al. [92], Dewey et al. [41], Midtgaard et al. [83], Austin et al. [6], Yoshikawa et al. [125], Zhao et al. [132], Ching and Katz [32], Kalinov et al. [64, 65] , Zhang et al. [131], Patra and Pradel [93], Bastani et al. [8]
Program Mutation (Section 3.4)	Semantics-Preserving Mutation (Section 3.4.1)	Le et al. [72], Le et al. [73], Lidbury et al. [77], Sun et al. [111], Donaldson et al. [42],
	Non-Semantics-Preserving Mutation (Section 3.4.2)	Nagai et al. [88], Chen et al. [31], Holler et al. [58], Garoche et al. [48], Groce et al. [49],

papers mainly focus on constructing test programs and do not address other research challenges. Meanwhile, these survey papers were completed 10–20 years ago, which may have limited aids for current research on compiler testing. Therefore, we believe it is necessary to conduct a new survey of the field of compiler testing.

3 CONSTRUCTING TEST PROGRAMS

Testing of any kind of software requires test cases. In the realm of compiler testing, programs form part of the test cases. In this article, we call such programs *test programs*. Constructing test programs for testing compilers is not trivial. We describe the challenges faced when constructing test programs in Section 3.1.

Approaches to constructing test programs can be broadly classified into three categories. Table 1 gives an overview of the broad classification and also shows how each category can be further divided. Test programs are either manually written or constructed automatically. We explain the manual approaches in Section 3.2. The automated approaches either generate program fragments and concatenate them into test programs, or mutate existing programs. Section 3.3 gives details

Table 2. Summary of the Programming Languages Targeted by Different Test-program Construction Approaches

Language	Approach
C/C++	Eide and Regehr [47], Yang et al. [123], Nagai et al. [87], Nagai et al. [88], Lindig [78, 79], Groce et al. [51], Le et al. [72], Le et al. [73], Morisset et al. [85], Zhang et al. [131], Sun et al. [111], Amodio et al. [4], Alipour et al. [3]
JavaScript	Holler et al. [58], Groce et al. [49], Patra and Pradel [93], Bastani et al. [8]
PL/I	Hanford [52]
Ada	Duncan and Hutchison [45], Austin et al. [6], Mandl [80]
Fortran	Callahan et al. [17], Dongarra et al. [44], Burgess and Saidi [16]
Java	Sirer et al. [108], Boujarwah et al. [14], Yoshikawa [125], Chen et al. [31]
Algol	Houssais [61]
APL	Ching and Katz [32]
Arden	Wolf et al. [120]
Haskell	Palka et al. [92]
Lusture	Garoche et al. [48]
mpC	Kalinov et al. [64, 65]
OCaml	Midtgaard et al. [83]
Pascal	Burgess [34], Bazzichi and Spadafora [9]
PLZ/SYS	Bazzichi and Spadafora [9]
Python	Bastani et al. [8]
Ruby	Bastani et al. [8]
Rust	Dewey et al. [41]
Scala	Zhang et al. [131]
GLSL	Donaldson et al. [42]

on automated test program generation, and Section 3.4 discusses mutation-based test program construction approaches. Because automated approaches for test program construction can create a large number of test programs with little effort, they are widely used in compiler testing.

Table 2 gives an overview of the programming languages targeted by the various approaches discussed in this section. Overall, the approaches cover a wide range of languages, with C/C++ being the most popular language.

3.1 Challenges for Constructing Test Programs

Constructing test programs for testing compilers is challenging. In particular, it is difficult to construct test programs that are valid, diverse, and meet certain requirements for testing.

Validity of test programs. It is non-trivial to construct valid programs, because of the restrictions some languages have on using certain language constructs only in a certain way or in a certain context. Constructing invalid programs is of limited usefulness for testing compilers, since a program goes through multiple phases of processing by the compiler; if a compiler is presented with an invalid input program, then the program tends to get discarded in the initial stages of the processing. For example, if a constructed JavaScript program contains a `return` statement that is not inside a function, then the program is considered syntactically invalid. Such a program does not reach the code generation or the optimization phases of a JavaScript engine. In C, for example, generating a program with undefined behavior cannot be considered as a valid test program, since the result is invariably correct. Similarly, for many languages, an identifier must be declared before

being used in the program. Constructing a program while maintaining such restrictions is not always straightforward. For typed languages in particular, maintaining type constraints during test program construction is not easy, which often leads to the construction of invalid test programs.

Diversity of test programs. As with all test inputs, constructed test programs should be diverse. A syntactically diverse set of test programs will exercise different parts of the compiler and potentially increase code coverage of the compiler under test. This can potentially aid in uncovering bugs. In the realm of testing compilers, diversity metrics such as distance between test programs [30] have been proposed. Although desirable, it is difficult to construct a diverse set of test programs, since an increase in the number of syntactic language constructs can adversely affect the validity of the generated programs. For example, it might be desirable that a set of constructed C test programs contain variables and their corresponding operations using all available types. Unfortunately, it is not very straightforward for a test program constructor to do so. With the use of more features, the constructed test program becomes more prone to be generated as invalid. This is probably the reason many test program generators that we discuss later in Section 3.3 only generate programs using a subset of all available language features.

Specific requirements imposed by a testing method. The construction of test programs also becomes difficult when certain testing methods impose restrictions. For example, a common testing method is to compile a program with two compilers and compare the execution results. In such a case, the input program should be free of undefined behavior. However, if we just test for compiler crashes, the input program does not necessarily have to be free of undefined behavior. Constructing test programs that follow these restrictions or conformance in addition to the validity and diversity requirements can be challenging.

3.2 Manually Constructing Test Programs

Manually constructed test programs have been used since the early days of compiler testing. Such test programs can be effective in uncovering bugs, since the programs can be tuned to a particular need and they are often written to test newly implemented features.

Popular compilers, such as GCC [38], compiler infrastructures, such as LLVM [39], implementations of the Java platform, such as OpenJDK [40], and browsers, such as Chromium [37], use extensive manually written test suites to test their implementations. For example, GCC comes with several test suites for both the runtime libraries and the language front-ends. The documentation of OpenJDK,[6] GCC,[7] and Chromium[8] all have guides for developers on how a test case should be written. In addition, there are independently developed conformance test suites, such as the Plum Hall Validation Suite [62] for C and C++ and Test262 for ECMAScript [46].

Although popular in practice, there has been very limited academic work on manually constructing test programs for testing compilers. One exception is by Callahan et al. [17], who describe the manual construction of 100 Fortran loops for testing a vectorizing compiler. Such compilers should be able to determine if a loop can be expressed using hardware-supported vector operations, i.e., if a loop can be vectorized. The primary goal of the test programs manually constructed by Callahan et al. is to check whether the compiler vectorizes vectorizable loops rather than to test the correctness of the generated code. To this end, each of the manually written loop contains some code that can be vectorized by the compiler. When passing these loops to compilers, Callahan et al. find that some compilers miss vectorizable loop statements.

[6] http://openjdk.java.net/jtreg/writetests.html.
[7] https://gcc.gnu.org/wiki/HowToPrepareATestcase.
[8] https://www.chromium.org/chromium-os/testing/test-suites.

Another description of manually constructing test programs is by Dongarra et al. [44], who collect subroutines and loops that contain parallelism opportunities written by other developers. Additionally, they themselves also manually write programs containing parallelism opportunities. The end goal is to measure if Fortran compilers automatically parallelize the loops.

Wolf et al. [120] describe their experience of manually constructing test programs based on a natural language specification of the programming language. Their approach is to study the specification sentence-by-sentence, and to create a test program for every testable requirement given in the specification. Their work targets the Arden language, a domain-specific programming language to describe medical knowledge.

3.3 Test Program Generation

Although effective, writing test programs manually needs significant effort. As a result, there have been constant efforts towards designing automatic test program generators for testing compilers. These efforts can be broadly classified into three categories: *grammar-directed*, *grammar-aided,* and *other approaches*. We explain these approaches in the following:

3.3.1 Grammar-directed Approaches. Grammar-directed approaches for test program generation take a language grammar as their input and generate programs based on this grammar. Grammar-directed approaches are the first approaches proposed for automatic test program generation to test compilers. Given the context-free grammar of a language, it is natural to walk over its productions to generate strings of the language. To this end, Purdom [96] presents an approach to testing the correctness of parsers and grammars based on context-free grammars. The primary focus is to validate parsers that are automatically generated from context-free grammars and to find non-reduced grammars (where the grammar contains symbols that cannot be used for a sentence derivation). The generation starts from a unique start symbol and proceeds by applying left-right rewriting rules from the grammar. The approach uses some heuristics to generate short sentences by recursively going over the grammar. For testing parsers, it is desirable that the generated programs cover different states and transitions of the parser. Purdom evaluates his approach on automatically generated LALR(1) parsers and shows that in many cases, the generated test programs are able to cover multiple states and transitions of the parser.

Purdom's [96] test program generation based on context-free grammar faces shortcomings when the objective is to test all parts of a compiler. By only using a context-free grammar, it is difficult to express context-sensitive features of a language. Since the end goal of Purdom is to test parsers that do not expect semantically correct programs, this shortcoming does not affect the approach.

Approaches that focus on testing harder-to-reach parts of the compiler, instead of only the parser, use different ways to address context sensitivity during test program generation. The initial attempts at generating compilable programs based on language grammars tend to extend the context-free productions with context-sensitive features. The extended grammars form a family of grammars known as *two-level grammars* and are introduced by Adriaan van Wijngaarden to specify ALGOL 68 [118]. Generation of test programs for the goal of testing compilers use three particular two-level grammars: *W-grammars*, *attribute grammars,* and *affix grammars*. The following gives a brief introduction of these three variants of two-level grammars and also explains the approaches that use each of them. For a comprehensive description about these three types of grammars, readers are referred to a tutorial by Koster [69].

Affix grammar. Originally invented for linguistic applications [82], affix grammars present a way of extending context-free grammars with context-sensitive notions using affixes to the grammar

productions. In principle, affix grammars and attribute grammars that we present later only differ in terms of their formal notation [69]. Both extend the context-free grammar using parameters.

Inspired by affix grammars, Hanford [52] proposes an approach that uses an extended context-free grammar. The extended grammar has rules that have other rules known *syntax generators* attached to them. A syntax generator effectively acts as a working store. Consider the following simplified example from Reference [52]:

$$<\textit{label declaration}> \rightarrow <\textit{declaration identifier}>,$$
$$<\textit{label}> \rightarrow <<\textit{lambda}>>.$$

The first line above is the grammar rule,while the second line is the syntax generator. Informally, these rules mean that whenever an identifier (say, *a*) is declared, then the syntax generator gets activated and a rule called $<\textit{label}> \rightarrow a$ gets added to the context-free grammar. Such addition of rules to the context-free grammar effectively allows programs to be generated with some context-sensitive features, e.g., "use after declaration." For each non-terminal in the grammar, a pseudo-random decision is made to select another non-terminal or a terminal. Hanford implements the approach as a program called *syntax machine*, which generates syntactically valid test programs for a subset of PL/I.

Another approach using affix grammars is presented by Houssais [61], who uses these grammars to generate programs that test an ALGOL 68 implementation. This approach restricts the domain of the affixes to integers. The program generator creates fragments of the language and their necessary context by enumerating over each of the productions separately and then the resulting fragments together into the final test programs. Additionally, the approach also has some code appended to the grammar that aids in the generation of syntactically correct programs.

Attribute grammar. The work by Hanford [52] presented above provides the basic idea that an extended grammar may be used to generate test programs for testing compilers. Later, Duncan and Hutchison [45] demonstrate a test program generator using an attributed test grammar.

Attribute grammars are another way of extending context-free grammars by adding attributes to the grammar rules. Attribute grammars are introduced by Knuth [67], who extends context-free grammars to express semantics by appending additional information (attributes) to some of the productions. Knuth's formulation has two types of attributes: synthesized attributes and inherited attributes. When the grammar rules are expressed as parse trees, the values of the synthesized attributes depend on the attribute values of the children, whereas the attribute values of inherited attributes depend on the attribute values of the parents.

Duncan and Hutchison [45] present a general test grammar, which serves as a guideline of how and where attributes are to be added to a concrete context-free grammar of a language. As an example, they apply their test grammar to a subset of a context-free grammar for Ada and add attributes to the grammar productions. The attributes are non-negative integers, and they guide the generation of test programs in the sense that the values of the attributes determine which grammar productions will be used during generation. They show that their approach can be useful in testing the optimizer of the Ada compiler.

Burgess [34] presents an approach to testing Pascal compilers also based on attribute grammar. Later, Burgess and Saidi [16] extend it to check for optimization errors in two Fortran compilers. Similar to the previous work [34], they extend a grammar by adding attributes with the additional possibility of assigning weights to grammar productions and generate so-called self-checking test programs. The basic idea of self-checking test programs is to generate assertions along with the test programs, an idea inspired by a work of Bird and Munoz [12]. The approach uses many heuristics to generate test programs to test known optimizations applied by compilers. A tester may control the applications of such heuristics and may specify weights to the grammar productions.

An approach by Amodio et al. [4] trains a recurrent neural network to generate test programs. The model is trained to generate data that conform to a given grammar and also respect additional well-formed properties of the language, such as defining a variable before using it. Such well-formed properties as specified use the formalism of attribute grammars. To enable the neural network to learn such properties, the approach computes a context vector for each program element, which encodes, e.g., the set of variables that has already been defined.

W-grammar. Developed by Adriaan van Wijngaarden to specify ALGOL 68 [118], W-grammars have been the first kind of two-level grammars. A W-grammar consists of two context-free grammars. One of these context-free grammars is a meta grammar that is used to generate terminal symbols for the second grammar. The meta grammar can be thought of as a way to generalize the context-free grammar. Using techniques such as *consistent substitution*, where all occurrences of a nonterminal are replaced with the same expansion symbol, W-grammars are able to enforce context sensitivity.

Bazzichi and Spadafora [9] use a context-free grammar extended with a parameter, which effectively is a W-grammar. They call the new grammar a *context-free parametric* grammar. The extension augments a parameter to some of the nonterminal grammar productions. For this particular case, the parameter itself is a grammar that generates variables. Their objective is to generate both valid and invalid programs to test compilers. Similar to the work by Purdom [96], the generation algorithms are skewed towards short derivations to a terminal and towards using all productions of the grammar at least once. Bazzichi and Spadafora implement their approach for testing PLZ/SYS and Pascal compilers.

The previous grammar-directed approaches for generating test programs are either based on a complete language grammar or on a subset of the complete grammar. Zelenov et al. [127] propose an alternate approach to generating test programs based on a model of a grammar. Given a language grammar, they introduce grammar transformations to generate a restricted model of a language grammar. This model contains the minimal set of productions required to generate well-formed sentences of the target language. This generator contains an iterator, which generates such model representations, and a mapper, which maps those representations to valid language sentences. Based on the same generation approach, but specifically aimed at testing optimizations, Zelenov and Zelenova [126] build a model of the compiler's optimizer and generate optimization-targeted tests. The basic idea is to build models of optimizations performed by a compiler and then to generate tests that contain optimization opportunities.

Lindig [78, 79] presents another approach that directly iterates over the grammar productions. His approach, called *Quest*, uses custom grammar-like productions to generate random test programs for testing C compilers. The goal of the generation is to test if the parameters passed to a function is received unaltered. The generation is driven by a BNF-style production system that serves as a generator. Each generator can either generate a type or a value for a type or take other generators as inputs. Since the goal is to test if values passed to a function is carried forward unaltered, the test programs contain global variables, functions, and calls to the functions using the global variables. The functions contain assertions that check if the received parameters have the expected values.

3.3.2 Grammar-aided Approaches. Grammar-aided approaches take a grammar as an input and in addition use some heuristics to address context sensitivity. These approaches start with a template-like, fixed code fragment that acts as a placeholder and then utilize the grammar to generate the rest of the program. In addition, to guide the overall generation, some approaches also take a test driver file as an input, while others augment it into grammar productions.

Sirer et al. [108] propose one such grammar-aided approach to testing the Java virtual machine (JVM) by probabilistically iterating over the grammar productions. Given a grammar and a skeletal program called seed, the approach outputs self-checking test programs, i.e., programs with assertions to validate the correctness of the JVM. The input grammar specification contains the productions that need to be augmented with other information, e.g., a limit on the number of times a particular production can be used, and guard conditions that describe the context in which a production is applicable. The skeletal program contains annotations about the probability of each production, along with holes to be filled with code fragments generated by the expansions of the productions.

Yang et al. [123] propose a grammar-aided tool, called Csmith, that creates C test programs. The tool is based on Randprog [47] (discussed in Section 3.3.3). In addition to functions, global and local variables, `const` and `volatile` variables, all of which exist in Randprog, Csmith generates programs that contain control flow statements, structs, arrays, and most kinds of C expressions. The approach uses complex heuristics to avoid generating C programs that have undefined behavior as well as that depend on unspecified behavior. The generation starts by creating a `main` function and a set of struct type declarations, each of which contains a random number of member variables of randomly decided types. Using the `main` function as the starting point, the rest of the C code is generated based on a subset of the C grammar. Depending upon the current state of the generation, Csmith chooses an allowable production based on a probability table and a filter function. The filter function enforces context sensitivity. During generation, Csmith performs certain safety checks and creates a code fragment only if all safety checks pass.

Several variants of the Csmith test program generator have been proposed. Morisset et al. [85] modify Csmith to generate programs that test C/C++ concurrency bugs. Their altered version adds support for mutexes and atomic variables, as well as system calls to lock and unlock the mutex variables. Also, Lidbury et al. [77] build a test program generator for OpenCL compilers based on Csmith, which is called CLsmith. CLsmith has six modes in total, and it generates different types of OpenCL kernels (i.e., test programs) under different modes. For example, in the VECTOR mode, CLsmith extends Csmith by introducing the capability to generate variables and expressions with vector types and exercising the rich set of vector operations available in OpenCL [77].

Instead of extending Csmith with additional language features, swarm testing [51] restricts the set of language features available for generating a specific program. For example, swarm testing may configure Csmith to not use any arrays for some test programs and not use any pointers for some other programs. The intuition of this approach is that some bugs are more likely to be covered by intensively using a subset of all language features instead of equally distributing the testing effort across all features. Later, Alipour et al. [3] further propose directed swarm testing, which aims to generate test programs focusing on only part of a compiler through tuning the set of available language features. More specifically, by analyzing statistical data on past testing results, directed swarm testing configures Csmith on a set of C language features to generate test programs with higher probability to cover a specific part of a compiler.

Furthermore, Chen et al. [27] propose an approach, called *HiCOND*, to finding a set of test configurations (which can control the language features of the generated test programs) for test-program generators (e.g., Csmith) to generate bug-revealing and diverse test programs. More specifically, *HiCOND* first infers the range of each configuration option where the bug-revealing test programs are more likely to be generated based on historical data and then identifies a set of test configurations that can generate diverse test programs via a search method (i.e., particle swarm optimization).

Instead of generating or hard coding the placeholder code as in the previous approaches, Holler et al. [58] use real-world programs as placeholders. They take these programs from a corpus and

replace some parts of the program with randomly generated code fragments. Code fragments get generated by iterating over the grammar productions and by taking a random decision whenever multiple choices are available for a particular production. After a maximum number of iterations, the remaining nonterminals are always replaced by terminals. These replacement terminals are again taken from a large code corpus. Additionally, the approach uses some heuristics, such as renaming of identifiers, to fit the generated code fragments into the target or placeholder program. The overall approach is a combination of both generation and mutation, and we explain it further in Section 3.4.

All previous grammar-aided approaches start with a given placeholder input. In contrast, Boujarwah et al. [14] first generate a test program and then generate additional code to obtain a semantically correct test program. According to a context-free grammar, their approach tests certain semantic features of Java compilers. They first decide upon which semantic features of the language they want to test and then use the context-free grammar, together with a test driver file, to guide the generation. The test driver file contains information about the language construct to generate, i.e., how many occurrences of each such language construct to generate, and the order in which they will get generated. For example, if during generation, Bourjarwah et al. decide to generate test programs for testing loops, then the test driver may contain which kinds of loops (for, while, do-while) should get generated and the number of them. Once a language construct has been generated, the approach generates the required context for the generated code. For example, they generate the required type declarations or add necessary imports. The iteration over the context-free grammar is based on Purdom's algorithm [96] discussed earlier (Section 3.3.1).

3.3.3 Other Approaches. Not all approaches for automatic test program generation are built on a grammar of the targeted programming language. The following discusses a set of approaches that do not use a grammar directly as an input. Many of them generate test programs based on pre-defined code templates that specify a skeleton for test programs, which is then filled with additional code snippets.

Berry [11] proposes test program generation for compilers based on the frequency of language features used by real programmers. To this end, they collect statistics of how language features are used and then design test programs based on frequently used features. The insight behind the approach is that features with higher usage frequency are more likely to be used by a compiler during its practical usage, and hence also more likely to trigger a bug. The approach generates different hard-coded language snippets based on the collected statistics.

Mandl [80] is an approach to avoiding the generation of duplicated elements in programs. It uses a unique approach called *orthogonal latin squares* to generating test programs for validating an Ada compiler. A latin square is a square matrix where each row and column contains an element exactly once, i.e., all elements of the matrix are unique. Two matrices are orthogonal latin squares if combining them creates another latin square. Mandl represents test templates as orthogonal latin squares and generates test programs by replacing elements of a row from the template matrix with allowable values. Using this latin square representation helps the approach to creating a test template with different configurations. Each configuration can generate multiple unique test programs. As an example, suppose the goal is to generate arithmetic expressions. Given the operands of the expression, the test template matrix could contain the operators of arithmetic expressions. Going through each row of the template matrix while using the operands, the approach generates unique arithmetic expressions.

The Csmith approach explained in Section 3.3.2 is an extension of a non-grammar-based approach called Randprog, which is introduced by Eide and Regehr [47]. As suggested by the name, Randprog generates random C programs to find miscompilations of C's `volatile` qualifier. Each

program generated by Randprog is hard-coded to first contain a number of randomly initialized global variables that are either `const` or `volatile`. The program then contains functions that declare local variables and that contain expressions using global and local variables. Finally, to make the programs executable, each program contains a `main` function.

An approach similar to Randprog is proposed by Nagai et al. [87]. They test code optimization for arithmetic expressions in C compilers. Each generated test program contains initialized global and local variables along with arithmetic expressions that use random operators and variables. The generation uses heuristics to avoid generating undefined behavior. During generation, the approach precomputes the result of each expression and inserts a runtime check that compares the precomputed result with the actual result. In a follow-up work, Nagai et al. [88] improve upon their work by mutating arithmetic expression, which we explain in Section 3.4.

Randprog [47] and the approach by Nagai et al. [88] start by generating fixed language constructs and then randomly generate the rest of the program. In contrast, Palka et al. [92], similar to some grammar-aided approaches (Section 3.3.2), start with a placeholder and expand the placeholder with random Haskell terms. In addition to the placeholder, they take type rules written in judgment proposition notation as inputs and generate well-typed Haskell terms. Each type rule has an associated environment that contains a list of identifiers and their corresponding types. For each type rule, the approach first recursively generates the terms present in the premise and then generates the terms in the consequence. Instead of directly using the type rules, Dewey et al. [41] present a fuzzing technique using *constraint logic programming (CLP)*. The type judgments are written in CLP specifications and used as an input to CLP engine for generation. They extend swarm testing ideas to find defects in the Rust type checker; i.e., instead of creating one generator that encompasses all of Rust, they create different generators that focus on different parts of the Rust type system.

The approach of Palka et al. [92] (presented in the previous paragraph) is later refined by Midtgaard et al. [83]. They observe that, in some languages, the evaluation of expressions depends on an unspecified evaluation order. For example, if the evaluation order is not specified, the outcome of the left-to-right evaluation of an expression can be different from the right-to-left evaluation. Since it is difficult to judge the correct outcome of an evaluation order-dependent expression, their approach tries to avoid creating such expressions. To this end, they refine the approach of Palka et al. [92] by introducing type and effect rules. During generation, the approach avoids the generation of programs that depend on the evaluation order using some predefined rules.

Instead of hard-coding the features of the generated test programs in the generator, some approaches use configuration or template files that a user can control. Austin et al. [6] present such a program generator for testing Ada compilers. The approach generates complex Ada expressions in a recursive descent manner by taking random decisions for alternative syntactic decisions. The generator is configurable in the sense that a tester can specify, e.g., the size of the integers produced or the seed used for random decision.

Yoshikawa et al. [125] present a similar configuration-driven generation approach with the knowledge of syntax and the application of heuristics. The generation starts by generating a random number of Java classes having acyclic parenthood relationships. This is followed by the generation of fields for each class followed by generation of methods. The methods are then filled with control flow information. For example, method invocations are added randomly into each method body. These additions are made with some constraints to avoid certain behaviors such as infinite method calls.

Zhao et al. [132] present another configuration-driven approach of test program generation that targets at testing compiler optimization. The approach takes a test configuration file as an input, which specifies, e.g., which variable type should be generated, what operators need to be

generated, and how many branches and loops are used in the test programs. Moreover, a user can configure which optimization to be tested. The optimizations to be tested are specified as temporal logic formulas, which in turn are converted into graph structures. The graphs are then converted to templates whose expansion leads to the generated programs.

Similar to Berry [11] (presented in the beginning of Section 3.3.3), Ching and Katz [32] propose an approach to testing an APL compiler using programs collected from real-world applications. Additionally, Ching and Katz [32] also propose generation of unit tests from templates. A template here represents a test with special symbols denoting functions and data types. During generation, these symbols are replaced with concrete values and the particular function being tested. The functions here are built-in functions of APL.

In addition to a template, an approach by Kalinov et al. [64, 65] also takes an input expression for test program generation that they use to find bugs in mpC, a parallel language compiler. The template contains a set of mpC operators, while the expression is a valid expression called seed expression provided by the testers. Starting from this seed expression, the approach generates multiple variant expressions by using the operands from the seed expression and operators from the template. The mpC language is specified as a visual formalism called Montages [70] that can express the syntax as well as the execution behavior. Kalinov et al. [64, 65] use this specification to filter out positive and negative examples during test program generation.

Zhang et al. [131] propose skeletal program enumeration. Given a program skeleton, i.e., source code with holes to be filled with variable names, the approach exhaustively enumerates all possible variable usage patterns. Because different assignments of holes to variable names may be equivalent, the approach enumerates only one program out of a set of equivalent programs under alpha renaming, i.e., under a consistent renaming of variables names.

Following the recent trend of using machine learning on software artifacts, e.g., source code, several approaches for learning-based generation of test programs have been proposed. They all share the basic idea of learning a model from a corpus of code examples and to then use this model to generate additional test programs. The approaches differ in the kinds of models they use and in the kinds of program properties they are targeting, as explained in the following. We have already introduced one such approach by Amodio et al. [4] in Section 3.3.1, who train a recurrent neural network to generate test programs. Another learning-based approach called *TreeFuzz* [93] learns a set of probabilistic generative models of tree-shaped data, such as programs represented by an AST. The models address both syntactic and semantic properties of programs, e.g., by learning what children nodes a particular node typically has or by learning definition-use-like relationships between occurrences of the same variable. The approach has been used for testing JavaScript engines.

Bastani et al. [8] propose to learn a grammar based on examples of data accepted by the grammar and on black-box access to a parser for the grammar. The approach iteratively constructs a grammar that accepts an increasingly general language, starting by synthesizing regular expressions and by then generalizing the regular expressions into a context-free grammar. Once the grammar is learned, it can be used to sample new test programs. The approach has been used for a variety of data formats, including test programs in Python, Ruby, and JavaScript.

3.4 Program Mutation

Instead of generating complete programs from scratch, the main idea of program mutation is to modify parts of an existing test program. However, it is possible that the existing program itself is generated using approaches presented in Section 3.3. It is interesting to note that most program mutation–based approaches for testing compilers are a result of very recent research efforts in the past decade. These research efforts are driven by the success of Csmith [123] and in

many cases either mutate Csmith-generated programs or build upon the shortcomings of Csmith. Furthermore, it is also interesting to note that a large portion of the mutation approaches find bugs in the optimization phase of the compiler. The reason for this is that, by code mutations and by complicating control flow, the approaches provoke the optimizer. Overall, the mutation approaches to constructing test programs can be classified into two categories. One category is based on semantics-preserving mutation (Section 3.4.1), and the second category is based on mutations that do not try to preserve the semantics of a program (Section 3.4.2).

3.4.1 Semantics-preserving Mutation. The main idea of semantics-preserving mutation is to mutate without changing the behavior of the program. Almost all semantics-preserving mutations are based on the general idea of *equivalence modulo inputs* (EMI) [72]. Informally, two test programs written in the same programming language are equivalent to each other under a set of inputs, if for each input of the set their behaviors are the same. Actually, EMI makes greater contribution to test oracles in compiler testing, and thus more details about it are presented in Section 4.2. Mutation approaches leverage the general idea of EMI by mutating programs to their semantic equivalences, and these semantic-preserving mutants can also be regarded as a type of test program.

Based on the general idea of EMI, Le et al. [72] propose *Orion* to validate C compilers by randomly mutating non-executed parts of code to create test programs. The premise of *Orion* is mutating non-executed part of the code should not alter the behavior of the program, and a diverging behavior can potentially be due to a bug. Later, Le et al. [73] extend *Orion*, called *Athena*, and instead of blind random mutations in *Orion* adopt a guided mutation strategy. Given a program, they mutate the non-executed parts of the program with the objective to generate a mutated program having a large distance with the original program. The distance is calculated based on control-flow graph (CFG) nodes, the distance between the CFG edges, and the program sizes. To guide the mutation, *Athena* uses Markov Chain Monte Carlo (MCMC) sampling that selects mutated programs with large distances. In contrast to *Orion*, where the only mutation operation was to delete non-executed code, they additionally insert code into the non-executed parts of the program. Both of these approaches use Csmith as the source program generator on which the mutations are performed.

Lidbury et al. [77] present fuzzing of OpenCL compilers also using the basic idea of EMI. They modify Csmith to generate programs suitable for OpenCL compilers and perform semantics-preserving code mutations. The mutation operation is the insertion of code known to be dead-by-construction at random locations of the original program.

The code mutation strategies adopted by Le et al. [72, 73] and Lidbury et al. [77] either delete or insert dead code. In comparison, Sun et al. [111] present *Hermes*, including some novel EMI-based mutation strategies. In addition to dead code, *Hermes* mutates live code or the code that gets executed. The mutation strategies adopted by *Hermes* involve insertion of code blocks where the conditional predicate always evaluate to false, wrapping an always true code block around live code and insertion of side-effects-free self-constructed live code. A similar mutation approach has also been proposed by Donaldson et al. [42], who apply semantic-preserving transformations to test graphics shader compilers. Some of the code transformations applied by them are similar to the mutation operations of Sun et al. [111]. In addition, they also apply code transformations such as mutation of numeric and Boolean expressions.

3.4.2 Non-semantics-preserving Mutation. In addition to semantics-preserving mutations, there exist approaches that mutate programs without keeping the same semantics. The main motivation of mutation for such approaches is to make a program suitable for testing compilers, e.g., by avoiding undefined behaviors or by creating more diverse test programs.

To this end, Nagai et al. [88] extend their previous work of generating random arithmetic expressions [87] (Section 3.3) using non-semantics-preserving mutation of the generated expressions. In their previous work [87], Nagai et al. avoid generating long arithmetic expression with the assumption that longer expressions are more susceptible to induce undefined behaviors. They improve upon this and avoid undefined behaviors in long expressions by mutating the undefined-behavior-producing expression with some heuristics and apply it for testing C compilers. The heuristics are, for example, flipping an operation or by inserting an operation. As a result, they are able to generate larger arithmetic expressions and are able to find errors in C compilers.

The main idea of Chen et al. [31] for testing JVM implementations, is similar to that of Le et al. [73] (Section 3.4.1), i.e., instead of blind mutations, performing MCMC (Markov Chain Monte Carlo) sampling. The objective is to select mutations that have larger possibilities to trigger compiler bugs. More specifically, to test JVM implementations, Chen et al. [31] mutate class files with a wide range of mutation operations. They have implemented many mutation operations, such as inserting/deleting methods into/from class.

Holler et al. [58] present a mutation-based approach called *LangFuzz* that finds bugs in the JavaScript interpreter of Mozilla Firefox. *LangFuzz* contains two phases: learning and mutation. In the first phase, it learns a large pool of code fragments by processing a set of sample input files using a parser. These are actually non-terminal expansions of the grammar. In the next phase, they parse a target program and replace some randomly chosen non-terminals with the expansions of the same type from the learned pool. For some cases, instead of choosing the replacement from the pool, they generate the replacement using a grammar. It is possible that the replacement fragments do not fit into the target program and they use some heuristics to fix this. For example, if the replacement fragment contains identifiers that are not declared in the target program, the target program might crash. To mitigate such situations, they rename all identifiers occurring in the replacement code fragment with some identifiers occurring somewhere in the target program.

Certain mutation approaches take a complete test suite as input and mutate with varying goals in mind. To this end, Garoche et al. [48] present a mutation approach with the goal of producing more failure-inducing programs using existing test suites. The mutation approach does not alter the control flow or the overall semantic structure, albeit changing the semantics of entire programs. The mutation operations are, for example, replacement of arithmetic or Boolean operations and replacement of constants with others. However, Groce et al. [49] mutate test programs with the goal of still maintaining certain properties. As an example goal, they successfully reduce the test suite of Spidermonkey while retaining the statement coverage. The overall mutation operation is a generalization of delta debugging [128] introduced by Zeller. Groce et al. call their reduction approach *cause reduction* and in addition to maintaining coverage, can also be used to keep other properties, such as maintaining the same failure-inducing test programs in the test suite.

4 TEST ORACLES

As any testing activity, compiler testing must address the test-oracle problem, i.e., to determine whether a given test program exposes any undesired behavior. To address this challenge, several approaches have already been proposed in the literature. We categorize these approaches into two groups: differential testing [81] (presented in Section 4.1) and metamorphic testing [29] (presented in Section 4.2).

4.1 Differential Testing for Compilers

To solve the test-oracle problem for complex software such as compilers, McKeeman et al. [81] propose the concept of *differential testing*. In general, differential testing for compilers needs at least two compilers that are designed and implemented based on the same specification, and then

Table 3. Usage of the Three Differential-testing Strategies in Compiler Testing

Paper	Cross-compiler	Cross-optimization	Cross-version
Sheridan [107]	✓		
Ofenbeck et al. [90]	✓		
Sassa and Sudosa [104]		✓	
Morisset et al. [85]		✓	
Le et al. [74]		✓	
Béra et al. [10]		✓	
Hawblitzel et al. [54]		✓	✓
Chen et al. [31]	✓		✓
Sun et al. [110]	✓	✓	✓

compares the results from these comparable compilers to determine whether compiler bugs are detected. To select the implementations to compare, there are several variants of differential testing for compilers. In particular, Table 3 summarizes the usage of three widely used differential-testing strategies in compiler testing.

- **Cross-compiler strategy:** Detect compiler bugs by comparing results produced by *different compilers*. This strategy is the most general concept in differential testing for compilers.
- **Cross-optimization strategy:** Detect compiler bugs by comparing results produced using *different optimizations* implemented in a single compiler. This strategy is the most widely used strategy in the existing compiler-testing research.
- **Cross-version strategy:** Detect compiler bugs by comparing results produced by *different versions* of a single compiler.

Sheridan [107] uses the cross-compiler strategy to test a C99 compiler, the PalmSource Cobalt ARM C/C++ embedded cross-compiler. In this work, Sheridan compares the output of the compiler under test and that of the preexisting tools to detect compiler bugs. In particular, they use the GNU C Compiler in C99 mode and the ARM ADS assembler as the preexisting tools. The insight as to why they utilize this strategy includes three points: First, different compilers for the same programming language are expected to produce the same output for the same input. Second, if the input can trigger a bug, different compilers seldom expose the same bug and produce the same buggy output under the same input due to the differences between their implementations. Third, if two compilers produce different outputs under the same input, one of them must contain a bug.

Ofenbeck et al. [90] propose to detect compiler bugs by taking random instances of an IR (intermediate representation) as inputs through differential testing, which is called *RandIR*. They target at vanilla Scala code in their study. When using *RandIR*, the users should give the grammar of the code that is represented by the IR [90]. Actually, the grammar is a collection of typed functions/operations. More specifically, *RandIR* first randomly constitutes the operations provided by users and records the information in a typed dependency graph. Then, it translates the constituted operations to regular Scala functions. To conduct differential testing, it still requires another regular Scala program. Here, the program can be produced based on the typed dependency graph or another compiler pipeline that transforms IRs to Scala functions. That is, they use the cross-compiler strategy for differential testing.

Sassa and Sudosa [104] use the cross-optimization strategy to test optimizers of a compiler. In particular, their approach detects compiler bugs by comparing traces of important values before and after optimizing the given test program. If the traces are different, then it means that a bug in the optimizer is detected. To conduct such a comparison, the first step is to determine which

values of variables should be compared. The second step is to determine when during the program execution the comparison should be conducted. The final step is to conduct the comparison. In particular, the comparison on values of variables is based on the traces of variable information, including basic block number, instruction number in the basic block, variable name, and the value of variables.

Morisset et al. [85] use the cross-optimization strategy to detect concurrent bugs in C11/C++11 compilers. Since concurrent test programs are not deterministic and compiler optimizations can compile away non-determinism, it is essential to ensure that all the behaviors of an executable produced by a compiler are allowed by the source-program semantics. To conduct differential testing for concurrent compiler bugs, they must assume that the sequential code that is optimized by C11/C++11 compilers can be run in any concurrent context. In particular, there is one constraint, which is that the test program is well-defined and can only apply sound optimizations for the concurrency model. More specifically, this work presents the correctness of the criteria for sound optimizations in the C11/C++11 model. Based on the theory of sound optimizations, they develop a tool, called *cmmtest*, to conduct differential testing of concurrent compiler bugs. The tool compiles the same test program using the compiler with optimizations under test and the compiler without turning on any optimization, and then records their memory traces (i.e., all memory accesses to global variables and synchronizations). Finally, it compares the recorded traces to determine whether a compiler bug is detected. In particular, it checks whether the trace from the compiler with optimizations under test can be transformed from the trace of the reference compiler by conducting some transformation rules, including the valid elimination rule, the reordering rule, and the introduction rule.

Le et al. [74] use the cross-optimization strategy to test link-time optimization (LTO) of compilers and develop a tool called *Proteus*. More specifically, *Proteus* compiles a test program in three different ways. The first way is that, the test program is directly compiled by the compiler under test without turning on LTO. The second way is that the test program is compiled by the compiler under test with turning on LTO and various other optimizations. The last way is that the test program is first split into a set of compilation units, and then the set of compilation units are separately compiled by the compiler under test under various optimizations and linked with turning on LTO. The results produced in the above three different ways should be the same. Otherwise, there is a compiler bug that is detected.

Béra et al. [10] use a variant of the cross-optimization strategy to test dynamic deoptimization of bytecode-to-bytecode JIT compilers. They compare the abstract stack of deoptimization and that of non-optimization to detect bugs. In particular, they apply symbolic execution on the bytecode produced by the compilation with optimizations and that produced by the compilation without optimizations. During the process of symbolic execution, when coming across a point where the dynamic deoptimization can be applied, they stop symbolic execution and then compare the stack of abstract values of deoptimization and that of non-optimization to guarantee the correctness of deoptimization for each value.

Some work uses multiple test oracle strategies at the same time. Hawblitzel et al. [54] propose to detect compiler bugs by comparing assembly language outputs of (1) multiple versions of a compiler (cross-version strategy) and (2) a compiler with different optimization levels (cross-optimizations strategy). Besides, this work also uses other strategies, including *cross-architecture strategy,* in which compiler bugs are detected by comparing results produced on different architectures for a compiler (i.e., ARM and x86); and *cross-scenario strategy,* in which compiler bugs are detected by comparing results produced in different compilation scenarios of a compiler (i.e., Just-In-Time and Machine Dependent Intermediate Language). During comparison, their proposed tool proves the equivalence between assembly language programs using the symbolic differencing tool

SymDiff, the program verifier *Boogie*, and the automated theorem prover *Z3*. Since an assembly language program consists of a collection of compiled methods, their tool first converts each of these methods into a procedure in the Boogie language. Then, it uses *SymDiff* to integrate converted methods into a single block of Boogie code. Next, it uses the *Boogie* program verifier to convert the assertions into verification conditions. Finally, it uses *Z3* to prove whether these verification conditions are valid. To reduce false alarms, their tool automatically produces counterexample traces that show values causing different behaviors in the two assembly language programs.

Chen et al. [31] propose to test JVM implementations via differential testing, focusing on the startup processes of JVMs. A JVM startup process includes four steps, i.e., loading, linking, initializing, and invoking classes. During the process of differential testing, their approach uses cross-compiler strategy and cross-version strategy to detect JVM discrepancies. More specifically, they propose a coverage-directed fuzzing approach, called *classfuzz*, to generate representative classfiles (referring to those that are likely to be distinct) for differential testing.

Sun et al. [110] propose to detect incorrect compiler warnings via differential testing. They use all the three strategies for differential testing, i.e., cross-compiler strategy, cross-version strategy, and cross-optimization strategy. Their approach first randomly generates test programs to make compilers emit various compiler warnings. Their program generation approach is based on two observations from historical warning bugs. First, most historical bugs are irrelevant to the bodies in conditional statements. Second, most historical bugs do not occur at the regions of obviously dead code. Then, since the warnings emitted by different compilers, different versions of one compiler, and different optimizations of one compiler version are described differently using the natural language, their approach conducts the alignment for these warnings. In particular, their approach extracts the warning elements that can be recognized by computers from warning descriptions for alignment. Finally, their approach identifies inconsistencies as potential warning bugs.

Besides, Kitaura and Ishiura [66] propose to detect performance bugs via differential testing. They use both cross-compiler strategy and cross-version strategy. Their approach is based on mixed static and dynamic code comparison. In the static step, it first compares the assembly codes produced from a given test program under two different compilers/versions to detect a code difference, and then reduces the test program to isolate the difference. In the dynamic step, it executes the codes produced from the reduced test program under two different compilers/versions to compare their actual execution time. For the performance of compilers, there is also some research on finding missed compiler optimizations (i.e., optimizations performed by one compiler but missed by another compiler) [7, 53, 86]. For example, Barany [7] uses differential testing (i.e., cross-compiler strategy) to find missed optimizations in C compilers. In particular, they develop a tool to statically compare the binary codes produced from a test program under two compilers.

4.2 Metamorphic Testing for Compilers

Metamorphic testing [29] is another popular approach for addressing the test-oracle problem. The core idea of metamorphic testing is to construct *metamorphic relations*, which specify how particular changes to the input of the project under test would change the output. For example, when testing the *sine* function, it is difficult to determine the expected output of $sin(1)$. However, the mathematical property of the *sine* function, i.e., $sin(x) = sin(\pi - x)$, can help test $sin(x)$. In other words, we can test whether $sin(1) = sin(\pi - 1)$ to facilitate the testing of the *sine* function.

To apply metamorphic testing to compilers, several metamorphic relations have been proposed, as summarized in Table 4. In particular, the most widely adopted metamorphic relations are the equivalence relations that establish that two programs are equivalent under some assumptions. The following discusses the approaches summarized in Table 4 in more detail.

Table 4. Metamorphic Relations Used in Compiler Testing

Paper	Metamorphic relation	How to construct metamorphic relations
Tao et al. [114]	Equivalence relation	Constructing equivalent expressions, assignment blocks, and submodules
Le et al. [72]	Equivalence relation under a given set of test inputs	Deleting code in the dead regions under the set of test inputs
Le et al. [73]	Equivalence relation under a given set of test inputs	Deleting and inserting code in the dead regions under the set of test inputs
Sun et al. [111]	Equivalence relation under a given set of test inputs	Inserting code in both the live and dead regions by synthesizing valid semantic-preserving code snippets under the set of test inputs
Donaldson and Lascu [43]	Equivalence relation	Injecting dead code into test programs
Nakamura and Ishiura [89]	Equivalence relation	Applying a set of equivalent transformation rules on test programs
Donaldson et al. [42]	Equivalence relation	Applying a set of (essentially) semantics-preserving transformations on high-value graphics shaders
Samet [99–101]	Equivalence relation	Converting a source program and the object program into an intermediate representation, respectively

Tao et al. [114] develop a testing tool, called *Mettoc*, to test compilers via metamorphic testing, where they consider the *equivalence-preservation relation* as the metamorphic relation. *Mettoc* generates at least two equivalent test programs and then uses the compiler under test to compile them to produce executables. After running these executables, producing different results means that a bug is detected. Here, *Mettoc* generates equivalent test programs by constructing equivalent expressions, assignment blocks, and submodules. Here, "assignment block" refers to a compounded statement consisting of a sequence of assignments, and "submodule" refers to a compounded statement that may contain conditional structures. More specifically, *Mettoc* first constructs a general control flow graph, and each block node of the graph represents a submodule. Then, *Mettoc* uses those equivalent statements or expressions to fill in each block node. Finally, it traverses the filled graph to generate the equivalent test programs.

Le et al. [72] introduce the concept of Equivalence Modulo Inputs (EMI) to test compilers. Different from the used metamorphic relations in Tao et al. [114], this work adopts the *equivalence relation under a given set of test inputs* as the metamorphic relation. Given a test program and a set of test inputs of the test program, EMI first generates a series of equivalent variants with the original one under the given test inputs. Taking the original program and its equivalent variants as the inputs of a compiler, the compiler produces executables accordingly. Then, these executables should produce the same results when executing under the given test inputs. Otherwise, there is a compiler bug that is detected.

In particular, EMI is a general idea and has three instantiations, called *Orion* [72], *Athena* [73], and *Hermes* [111]. The difference among them is mainly the way of generating equivalent variants with a given test program under a set of test inputs. *Orion* generates equivalent variants by

randomly deleting non-executed statements under the given test inputs [72]. *Athena* utilizes the MCMC (Markov Chain Monte Carlo) algorithm to guide the generation process by introducing insertion operations in the non-executed regions besides deletion operations [73]. *Hermes* further introduces mutations in both live and dead regions by synthesizing valid semantic-preserving code snippets under the set of test inputs [111]. More details about the way of generating equivalent variants have been presented in Section 3.4.1.

Similar to the idea of EMI [72], Donaldson and Lascu [43] utilize metamorphic testing to test OpenGL compilers. This work proposes to inject dead code into existing test programs to construct equivalent program variants. The original test program and its equivalent program variants should produce the same results. Otherwise, a compiler bug is detected. Furthermore, Nakamura and Ishiura [89] propose to construct equivalent programs to test C compilers by designing a set of equivalent transformation rules on existing test programs.

Later, Donaldson et al. [42] propose to test graphics shader compilers via metamorphic testing. More specifically, they leverage existing graphics shaders of high value to create sets of semantically equivalent transformed shaders by applying a set of (essentially) semantics-preserving transformations. Here, a semantics-preserving transformation means that the transformation will have no impact on computation. The exception is the floating-point computation in which it is possible to induce slight changes at the transformation point. That is, it is different from the deterministic equivalence used in the existing work [72, 114].

Different from the metamorphic relations whose inputs are equivalent *source programs*, Samet [99–101] tests compilers based on the metamorphic relations whose inputs are *a source program* and *the object program* produced by a compiler under test. Their approach utilizes the intermediate representation (IR) to evaluate the equivalence between a source program and the object program produced by a compiler to detect compiler bugs. It first converts a source program into an IR, then converts the object program to the IR, and finally checks the equivalence between the two IRs. In particular, the IR of the object program is obtained by the process of symbolic interpretation.

5 OPTIMIZING THE TEST PROCESS

Typically, compilers are well tested and are widely used, and thus it is difficult to detect latent bugs in compilers. In fact, compiler testing approaches tend to take a long period of testing time to find a relatively small number of compiler bugs through running a lot of test programs [26, 73, 74, 123]. To address this efficiency problem, some optimization approaches for test-program execution have been proposed in the literature [19, 22, 23, 49, 122]. These optimization approaches can be divided into two types: test-program prioritization (presented in Section 5.1) and test-suite reduction (presented in Section 5.2).

5.1 Test-program Prioritization

In compiler testing, among all available test programs, only a small number can trigger bugs. Therefore, if we run these programs earlier, the efficiency of compiler testing can be improved. Test-program prioritization is a way to optimize the test-program execution order such that the programs with larger possibilities to trigger compiler bugs can be executed as early as possible [124].

Traditional test prioritization approaches [124] are infeasible for compiler testing, and the reasons are presented below. First, most traditional test prioritization approaches rely on code coverage (e.g., statement coverage). The coverage information can be collected in the regression testing scenario [98, 130]. However, the test programs for compiler testing tend to be randomly generated on the fly via automated program generators (e.g., Csmith [123]). Therefore, their coverage

information is not available before testing. That is, coverage-based test prioritization is not a good match for accelerating compiler testing. Second, there are also some other test prioritization approaches based on only test-input information [63, 115]. Unfortunately, an existing study [22] has evaluated the effectiveness of the state-of-the-art test-input–based test prioritization approach on accelerating compiler testing [63], and their results demonstrate that it cannot accelerate compiler testing, since the time required for prioritization is very long. The limited applicability of traditional test prioritization approaches motivates new work on test-program prioritization for compiler testing.

Chen et al. [23] propose a test-program prioritization approach by transforming each test program to a text-vector. Here, this approach considers three categories of bug-relevant tokens, i.e., statements, types and modifiers, and operators, for the transformation. Based on these text vectors, this approach then uses three strategies to prioritize these test programs: (1) The first one is to schedule test programs following the descending order of distances between text vectors and the origin vector (0,0,...,0), which is called the *greedy* strategy. (2) The second one is to schedule test programs by borrowing the strategy of *adaptive random* testing. More specifically, it selects the next test program, minimizing the distance to the already selected test programs. (3) The third one is to schedule test programs using a local beam search strategy.

Later, Chen et al. [22] propose the idea of "learning to test." Based on this idea, they implement an approach (called *LET*) to prioritizing test programs. The key insight of *LET* is that, if a test program contains certain features, then it can be hard to compile or optimize, and thus it has a larger possibility to trigger compiler bugs. LET first extracts two categories of features that are helpful to reveal compiler bugs to some degree. The first one is *existence features*, reflecting whether some types of elements are in the target test program. The second one is *usage features*, reflecting how these elements in the target test program are used. Based on the two categories of features, LET builds a *capability model* and a *time model*. The former aims to predict the bug-revealing probabilities for new test programs, while the latter aims to predict the execution time for new test programs. Based on both models, LET computes the bug-revealing probability per unit time for each new test program. Finally, LET schedules these new test programs according to the computed values.

Since the above proposed test-program prioritization approaches ignore the case in which different test programs may have the same test capabilities (i.e., testing the same functionalities of a compiler, even detecting the same bugs), such neglect may discount their acceleration effectiveness. To relieve this problem, Chen et al. [21, 28] propose an approach to distinguishing test programs with different test capabilities based on test coverage information. As discussed before, test-program coverage information in compiler testing is not available in advance, and thus he proposes to predict coverage statically based on test-program features, without test execution. Then, according to the statically predicted test coverage, this approach clusters test programs into different groups, each of which tends to have the similar test capability. Finally, this approach ranks test programs by iteratively enumerating each group.

5.2 Test-suite Reduction

Test-suite reduction is also a way to reduce testing costs [24]. It improves the efficiency of compiler testing by excluding redundant test programs. There are at least three approaches aimed at test-suite reduction for compilers. Two of them focus on retargeted compilers for embedded processors, which are developed by adapting existing compilers [19, 122].

Woo et al. [122] propose an approach to removing redundant test programs for retargeted compilers according to the IR-level coverage information. They work at this for three reasons: (1) different test programs at the source-code level are able to be mapped to the same program

at the IR level; (2) the machine code produced by compilers has the direct dependency on the IR; (3) the modifications of retargeted compilers extensively occur at the compiler back-ends during retargeting. This approach aims to acquire a test suite that has a smaller size but does not reduce the grammar coverage of the intermediate language [71]. More specifically, it assures that every preserved test program in the test suite should be able to cover at least one new grammar rule.

Chae et al. [19] further extend the above test-suite reduction approach [122]. First, they investigate a new kind of grammar coverage in their approach to test compilers such as n-state path coverage [64], making this approach more general. Second, they develop a fully automatic tool to reduce a test suite, which first generates a test suite according to the used coverage criteria and then reduces this test suite using the proposed reduction approach.

In addition to excluding redundant test programs from a test suite, Groce et al. [49] propose to reduce a test suite of a compiler by simplifying each test program but retaining all test programs. This approach is called *cause reduction. Cause reduction* is actually a generalization of delta debugging [128]. It simplifies each test program by maintaining test coverage.

6 POST-PROCESSING OF TEST RESULTS

Once compiler testing has found test programs that trigger a compiler bug, the next step is to understand and fix these bugs. To facilitate this task, several research efforts focus on *post-processing of test results*. We discuss these efforts in three groups: test program reduction (presented in Section 6.1), duplicated bug identification (presented in Section 6.2), and compiler bug debugging (presented in Section 6.3).

6.1 Test Program Reduction

Test programs tend to be large and complex. Therefore, an important step before reporting a compiler bug to a compiler developer is to produce a small test program that still triggers the compiler bug, i.e., *test program reduction*. This is because small test programs can facilitate the debugging of compiler bugs for developers. This post processing is also encouraged by compiler developers. For example, in the documentation of LLVM, there is the following declaration: "*...to narrow down the bug so that the person who fixes it will be able to find the problem more easily...*"

Delta debugging is an approach to determining the minimal set of failure-inducing changes in a faulty program in general [128]. Also, Zeller and Hildebrandt [129] apply delta debugging to simplify tests and utilize delta debugging to isolate the difference between passing tests and failing tests. Furthermore, there are various delta debugging algorithms proposed in the literature [5, 84]. However, they cannot effectively reduce *test programs of compilers* well, since they tend to get stuck in the local optima that are still too large. Besides, for some compilers such as C/C++ compilers, they often generate test programs with undefined behaviors, which are useless for those compilers.

To solve these problems, Regehr et al. [97] leverage domain-specific knowledge to solve the local optimal problem and avoid undefined behaviors directly during reduction. More specifically, they design and implement three reducers for test programs of C compilers as follows:

- The first one is called *Seq-Reduce* test-program reducer, which only works for test programs generated by Csmith. Since test programs generated by Csmith are determined by a sequence of integers produced by a pseudo-random number generator, the Seq-Reduce test-program reducer iteratively generates the variant that still triggers the bug but is smaller than the smallest variant produced previously, by randomly modifying the sequence.
- The second one is called *Fast-Reduce* test-program reducer, which also only works for test programs generated by Csmith. It supports a series of transformation rules such as dead-code elimination and exploiting path divergence. These rules are based on the information of both the static structures of the generated test program and its runtime behaviors.

- The third one is called *C-Reduce*, which works for any test program for C compilers. The C-Reduce test program reducer applies a set of pluggable transformations performing operations for reduction to a test program, until the test program cannot be reduced anymore (i.e., reaching a global fixpoint). It is the most effective one among the three reducers for reducing test programs triggering C compiler bugs. In particular, their experimental results demonstrate that the reduced test programs produced by C-Reduce are smaller than those produced by their other two reducers (i.e., Seq-Reduce and Fast-Reduce) over 25× on average.

Pflanzer et al. [94] further extend the C-Reduce test program reducer to OpenCL. That is, they provide an automated approach to reducing OpenCL test inputs triggering bugs. The key challenge is to detect undefined behaviors in an OpenCL kernel. To address this challenge, they build a new plugin (called *ShadowKeeper*) for *Oclgrind* [95], which is able to precisely detect accesses to data without initialization. In particular, the internal mechanics of the *ShadowKeeper* plugin borrow the ideas of the Memcheck plugin in Valgrind [106] and MemorySanitizer in Clang [109].

The above-introduced reducers are specific to some single kind of test inputs (e.g., C programs or OpenCL kernels). Herfert et al. [56] propose the generalized tree reduction algorithm, called GTR, to reduce any test inputs that are tree-structured, such as Python and JavaScript. This approach is independent of programming languages. Its input is a tree that has a property (e.g., to trigger a specific compiler bug) that should be preserved during reduction. Its output is a tree that has been reduced but still has the property. During the reduction process, GTR minimizes the whole tree by considering all nodes at a level, and then continues to the next level for further reduction. More specifically, GTR designs tree transformation rules and utilizes the delta debugging algorithm and a greedy algorithm for backtracking. The designed tree transformation rules are the key part of GTR, which performs reduction for a tree and then produces a new but smaller tree. In particular, GTR provides two transformation rules: (1) removing a node and all its children; (2) replacing a node with one of its children. Actually, GTR is easy to extend with additional transformation rules. Furthermore, their experimental study also demonstrates that GTR significantly outperforms the existing improved delta debugging algorithms [57, 84] for reducing tree-structured test inputs.

Sun et al. [113] propose another general framework, called *Perses*, for test program reduction. Besides the above-mentioned tree-structured test inputs, it can be used to reduce other structured test inputs, e.g., reducing structured text formats in the security domain. The key insight of *Perses* is to take the formal syntax of a programming language as a guide for reduction. It ensures to consider only smaller and syntactically valid program variants in each step of reduction so the syntactically invalid program variants can be avoided and the corresponding efforts can be saved.

Holmes et al. [60] define the problem of *slippage* in test input reduction (including test program reduction), which means that the bug triggered by a test input is different from that by the test input after reduction. Slippage may be harmful or beneficial, since the bug triggered by the reduced test input may be an already-known one or a new one. To avoid harmful slippage and induce beneficial slippage, they propose to produce a set of reduced test inputs for a given bug-triggering test input, instead of producing only one reduced test input. More specifically, they propose two approaches, *comb-block* and *multi-ddmin*, based on the delta debugging algorithm *ddmin* [128, 129]. The former blocks some components during delta debugging, while the latter randomizes the checking order for smaller test inputs during delta debugging to produce a set of different reduced test inputs.

Christi et al. [33] investigate the impact of delta debugging for test input reduction (including test program reduction) on spectrum-based bug localization that localizes bugs based on statistics of coverage status of each program element during failing and passing executions [1, 121]. The experimental results show the advantage of using reduced failing test inputs on spectrum-based

bug localization compared with using original failing test inputs, indicating that it is necessary to use delta debugging to reduce failing test inputs before spectrum-based bug localization.

6.2 Duplicated Bug Identification

Since random testing, or fuzzing, is one of the most important approaches to detecting compiler bugs, those fuzzers suffer from a serious problem: A fuzzer may produce a large number of bug-triggering test programs during the testing time of one night, and many of those test programs actually trigger the same compiler bug. Such redundant bug-triggering test programs enhance the debugging difficulty for developers, since compiler developers are expensive and limited in numbers. In particular, the existing work [30] reports that some industrial compiler developers stop using Csmith due to this problem. Therefore, it is necessary to identify such duplication before reporting compiler bugs.

Chen et al. [30] formulate this problem as *the fuzzer taming problem*, which is that, given a large number of bug-triggering test programs, it is essential to rank these test programs so the test programs triggering distinct bugs are ranked at the early positions in the list. Moreover, there is also an additional condition: If a test program triggering compiler bugs is marked as an undesirable one previously, the test program should be ranked at the late position in the list. To tame compiler fuzzers, Chen et al. [30] propose to distinguish test programs that trigger distinct compiler bugs through defining a set of distance functions to measure the similarity between test programs (based on a set of identified static and dynamic characteristics of test programs). It then ranks these test programs based on the furthest point first algorithm, which iteratively selects the next test program that has the maximum distance with the nearest one among all the existing selected test programs. The key insight of this approach is that, if two test programs have a farther distance, then they have a larger possibility to trigger two distinct bugs.

Holmes and Groce [59] further solve the fuzzer taming problem by proposing a mutation-based metric to measure the similarity between two bug-triggering test programs. The key insight is that, if two bug-triggering test programs become passing due to the same mutant of the compiler under test, then they are more likely to trigger the same compiler bug. More specifically, they first collect a set of mutants of the compiler under test and record which mutants make the test program passing for each bug-triggering test program. Then, they calculate the Jaccard distance between bug-triggering test programs based on the recorded information. Finally, they rank bug-triggering test programs based on the furthest point first algorithm like Chen et al. [30] so the test programs that are more likely to trigger different compiler bugs are ranked higher.

6.3 Compiler Bug Debugging

After receiving compiler bug reports, debugging these bugs is the next step, which is also important and challenging [102].

Some work focuses on finding the bug-triggering part of a test program and the bug-triggering condition to facilitate the debugging of compiler bugs. For example, Caron and Darnell [18] develop a tool called *Bugfind* to debug compiler bugs, which can isolate modules (i.e., files) that are not optimized correctly when multi-module projects are used as test programs. In particular, this tool targets at optimizing compilers, which have one or more levels of optimization. Compared to previously existing approaches, e.g., binary search by hand and semi-automated shell scripts, the *Bugfind* tool is automated and requires minimal human intervention. Moreover, it finds the failing modules in a minimal amount of time. To be specific, *Bugfind* first divides a failing multi-module test program into small files that can be compiled separately, and then turns up or down the optimization level for each file to find the files causing incorrect optimizations.

Whalley [119] proposes a tool, called *upoiso*, to automatically identify the first transformation within a function that causes incorrect results. In particular, this tool targets both optimization bugs and non-optimization bugs in the *upo* compiler system. For optimization bugs, it utilizes the binary search to identify the first transformation causing incorrect results by limiting the number of transformations applied to a specified function. For non-optimization bugs, it utilizes the binary search to identify the first function containing incorrect instructions by modifying the labels in the native assembly files generated by a native compiler.

Besides, some work focuses on providing more sufficient execution information to facilitate the debugging of compiler bugs. Sloane [102] proposes *Noosa* to debug the compilers that are generated using the Eli generation system. *Noosa* conducts the visualization for the compiler execution when processing test programs. The aim of *Noosa* is to make the debugging process conduct at the specification level. That is, it makes the implementation details of compilers be hidden, and thus the compiler execution can be understood even if there is no knowledge about it.

Hemmert et al. [55] propose to debug the SC (short for Sea Cucumber) synthesizing compiler at the source-code level. The synthesizing compilers make the debugging and verification of the operation of the hardware applications harder. The proposed debugger conducts the mapping between the executing circuit state and the source code. Such an approach not only utilizes the high efficiency of hardware, but also displays the messages of debugging at the original source code, which is helpful for users to conduct fast debugging of the circuit.

Ogata et al. [91] propose to debug a Just-In-Time (JIT) compiler by replaying the compilation based on two compilers. The approach is named *replay JIT compilation*. In particular, the first compiler used in the approach is the state-saving compiler, which is responsible to record all the runtime information in a normal compilation. The second compiler is the replaying compiler, which is responsible to replay the compilation in the debugging mode to output diagnostic information.

Chang et al. [20] propose to facilitate the debugging of native-code compilers in addition to bytecode compilers by conducting the assembly-language level type checking. More specifically, the assembly-language level checking is extended from the intermediate-language level checking. The extension additionally maintains a lattice of dependent types, which actually enhances the complexity of the approach. Even so, it provides the opportunity to debug native-code compilers through the checking technique.

Furthermore, Holmes and Groce [59] propose to utilize the mutation-based metric presented in Section 6.2 to help localize compiler bugs. The localization technique is called *Repair*. For a bug-triggering test program, *Repair* calculates a score for each mutant that makes the test program passing, and then ranks these mutants so the statements changed by the mutants ranked higher are more likely to be suspicious. The key insight is that, if a mutant can only make the bug-triggering test programs—which are similar to the given bug-triggering test program based on the mutation-based metric, passing—the mutant should be ranked higher. More specifically, the score of a mutant is calculated based on the maximum distance between the given bug-triggering test program and other bug-triggering test programs that the mutant makes passing.

Recently, Chen et al. [25] propose an approach, called *DiWi*, to localizing compiler bugs by transforming the problem of compiler bug localization into a search problem, i.e., searching for a set of effective witness test programs that are able to eliminate innocent compiler files from suspects (the compiler files involved when compiling the failing test program). More specifically, *DiWi* designs a set of witnessing mutation rules and proposes a heuristic-based search strategy to generate such a set of effective witness test programs based on a given failing test program. Then, *DiWi* isolates compiler bugs by comparing the coverage between the set of witness test programs and the given failing test program following the practice of spectrum-based fault localization [2].

7 EMPIRICAL STUDIES ON COMPILER TESTING

In addition to developing new approaches that address technical challenges, the compiler testing research field is benefiting from several empirical studies. These studies systematically explore compiler bugs and compiler testing approaches, providing insights on compiler testing.

Chen et al. [26] conduct a comparison study on three mainstream compiler testing approaches, including randomized differential testing (abbreviated as RDT), different optimization levels (abbreviated as DOL), and equivalence modulo inputs (abbreviated as EMI) [72]. Here, RDT refers to the approach testing compiler using the cross-compiler strategy, whereas DOL refers to that using cross-optimization strategy. To compare them precisely, they propose Correcting Commits, a novel measurement that searches for the commits correcting the detected bugs and uses such commit number to estimate the bug number. They prepare a fixed sequence of test programs generated by Csmith and apply them to GCC and LLVM, respectively, recording compiler bugs (measured by Correcting Commits) revealed by each approach during 90 hours. Following this experimental methodology, this study gets some findings. For example, DOL performs the most effective on the bugs related to compiler optimizations, RDT performs the most effective on the bugs not related to optimizations, and EMI complements them due to its effectiveness on detecting unique bugs. Moreover, the study investigates the factors impacting a compiler testing approach, including its efficiency, its used test oracle, and kind of test programs. They find each of the factors significantly impacts a compiler testing approach. In particular, the efficiency of the approach has the most impact, while the impact of the kind of test programs is the least. Besides, this study also discusses the combination of these approaches and suggests their usage order to be DOL, RDT, and EMI.

Lidbury et al. [77] perform an experimental study to investigate the effectiveness of RDT and EMI on a new application, i.e., OpenCL compilers. To apply RDT to OpenCL compilers, they build a tool called *CLsmith* using *Csmith* [123] to randomly generate test programs (i.e., OpenCL kernels) for OpenCL compilers. In particular, *CLsmith* provides several strategies for the random generation. Moreover, to apply EMI to OpenCL compilers, they propose to first inject dead code into OpenCL kernels and then apply the EMI technique (i.e., Orion [72]) to generate equivalent variants. Based on the two testing approaches, they apply inputs randomly generated by CLsmith to four versions of OpenCL in its 21 configurations (differ in devices and drivers), recording bugs revealed in a specified time-out. According to the empirical study, more than 50 OpenCL compiler bugs are identified and reported, most of which are in commercial implementations.

Sun et al. [112] conduct an empirical study to investigate the characteristics of compiler bugs, aiming to facilitate the understanding of compiler bugs. In particular, they study GCC revisions from August 1999 to October 2015 and LLVM revisions from October 2003 to October 2015 and choose the revision that is a fix to a bug based on its commit message. Through this process, this work collects 39,890 GCC bugs and 12,842 LLVM bugs. Based on the two bug repositories, this study investigates four important aspects of compiler bugs, including where the bugs occur, which characteristics the test programs triggering bugs and bug fixes have, how long the bugs can be detected and fixed, and what the cases of bug priorities are. Based on their results of the four aspects, they get some findings. First, the component processing C++ programs is the most buggy one, which should attract more research attention. Second, the test programs triggering bugs usually have a small size, indicating that we should produce small but complex test programs for compiler testing. Third, the fixes of bugs tend to involve only one file, and the size of these fixes tend to be small, indicating that we can conduct testing for a specific component. Finally, debugging the bugs usually takes a few months due to the complexity of compilers.

Groce et al. [50] conduct an empirical study to investigate the relationships between what a test program contains and what the compiler does in testing. They first define “triggers” and

"suppressors," where the former means that certain test-program features can make a test program *more* likely to explore some compiler behaviors (such as bugs and coverage entities) in testing, while the latter means that certain test-program features can make a test program *less* likely to explore some compiler behaviors. In particular, this study focuses on investigating how test-program features can suppress a compiler behavior, including the frequency and degree of "suppressors" in compiler testing. The study demonstrates that suppressors are quite common, and many features of test programs act as suppressors for some compiler behaviors in testing. They further discuss the causes and impacts of "suppressors." For example, some test-program features acting as suppressors have to be omitted in test programs to increase the probability of triggering some bugs.

8 OUTLOOK AND CHALLENGES FOR FUTURE WORK

Although significant progress has been made in the area of compiler testing, the problem is far from being solved. In this section, we highlight some challenges that future work may address.

Efficiency of Compiler Testing. Though some efforts have been made to enhance the efficiency of compiler testing, compiler testing is still a time-consuming task. Further efforts are still needed to enhance the efficiency of compiler testing.

One efficiency bottleneck is that a bug is often discovered multiple times by different test programs in compiler testing. This causes not only an efficiency problem but also a large amount of extra work for developers to review and classify the duplicated test programs. It would be desirable to have an approach that generates test programs triggering only new bugs, possibly by using the feedback from known bugs-inducing test programs. Chen et al. [23] have shown that by extracting feature vectors from existing test programs, better prioritization of test programs can be achieved. Such approaches may be integrated into the generation process to enhance test efficiency. Similarly, Chen et al. [22] have shown that learning from existing bug-inducing test programs could help us to recognize future bug-inducing test programs for test program prioritization. Such approaches may also be integrated into test generation to directly generate programs that have larger possibilities to trigger bugs.

Compiler Verification. Compiler verification is another family of attempts to eliminate bugs in compilers [36, 76]. However, due to the high cost of developing formal specifications and proofs, verification techniques usually can be applied only to a small part of compilers and guarantee a selection of properties. How testing can complement verification may be another research direction, where the focus could be the parts of a compiler that have not been verified.

Generalizability of Approaches. Though tremendous progress has been made in automating compiler testing, most of these efforts are specific to one particular programming language. In particular, many existing approaches are specific to C. For example, one of the most widely used tools for test program generation, Csmith [123], is designed only for generating C programs. Migrating such tools to other programming languages is not easy. Csmith has been carefully designed so no program triggering undefined behaviors will be generated, and a bunch of static analysis has been integrated into the generation process to prevent such programs. To migrate such tools to other languages, we need to carefully understand the rules for valid programs in other programming languages and design static analyses to best approximate such rules at generation time. These tasks require expert knowledge in program analysis and semantics and are not easy to perform. Therefore, future research could focus on providing approaches that can be generalized to different compilers: For example, by allowing easy specification of valid programs or by deriving specifications automatically from compilers, and deducing generation procedures from the specifications.

Recent work on learning-based test program generation [4, 8, 93], which can be easily applied to different programming languages, is the first step in that direction.

Another related problem is whether the current approaches can generalize to other software systems that also take programs as inputs, such as refactoring tools [35], debuggers [75, 116], static bug detectors, code search engines, and so on. More research is required to understand the difference between these tools and compilers and what approaches could be generalized.

Coverage of Bugs. Current automatic approaches for constructing test programs and test oracles can only cover a subset of all possible test programs and oracles, therefore covering only a subset of bugs among all compiler bugs. In test program generation, to generate only valid programs without undefined behaviors, tools like Csmith [123] rely on static analysis to approximate the boundary of a valid program, and thus cannot generate all valid programs. Also, some bugs may require invalid programs to trigger them. Similarly, the automatic oracles mainly rely on equivalence relations between programs/compilers, and thus cannot be used to discover bugs that require oracles beyond equivalence relations.

Therefore, future studies are needed to understand test programs and test oracles beyond the current spaces. While generating invalid programs is easy, understanding which invalid programs are likely to trigger bugs may not be easy. Programs with undefined behaviors or non-deterministic behaviors could also be taken into account. In the space of test oracles, we may consider relations that are not equivalence relations. For example, when optimizing the program size, the program size should become smaller than that before optimization.

Handling the Discovered Bugs. While a lot of research efforts have been put into discovering bugs, the processing of test results has received relatively little attention. Here, we highlight some challenges in bug handling.

One of the open challenges is which test programs should be brought to compiler developers. As mentioned before, existing research [30] has studied how to identify test programs that trigger the same compiler bug. However, even with these approaches, redundant test programs still exist and more studies are needed to understand whether they can be further reduced. A similar problem is the prioritization of bug reports. In early development stages of a compiler, a testing process may produce many bugs—how to prioritize the bugs and report to developers remain open problems.

Another challenge is the readability of test programs. While several attempts [97] have been made to reduce test programs, smaller programs do not necessarily mean better readability. How to model readability and how to achieve better readability are problems left to be solved.

Due to the complexity of compilers, fixing compiler bugs is not easy. Recently, significant progress has been made in automatic program repair. Can compiler bugs be repaired automatically? Can existing repair approaches be applied to compilers? Additional research is needed here to answer these questions.

Benchmarks. Currently, there is no established benchmark of compiler bugs for evaluating the effectiveness of compiler testing. Existing studies often spend huge efforts to discover bugs in existing compilers during their evaluation [22, 26]. Agreeing on a common benchmark could further promote the development of compiler testing research. A benchmark makes it much easier to measure the effectiveness of a newly proposed compiler testing approach.

9 CONCLUSION

This article provides a survey of approaches for testing compilers. Given the importance of compilers as a basic part of every developer's tool chain and the disastrous consequences of compiler bugs, testing compilers is an extremely important topic. Recent years have seen significant improvements and activities in the field of compiler testing. Our article enables interested outsiders

to obtain an overview of this thriving field and may enable experts to fill any gaps in their knowledge of the state-of-the-art. Based on our discussion of existing work, we conclude compiler testing has evolved into a mature field that has already made significant impacts on real-world compiler development.

Despite all the advances on compiler testing, there remain several interesting challenges to be addressed in the future, including how to generalize existing approaches and how to further improve both their effectiveness and efficiency. We hope that our survey allows researchers to make progress towards these goals.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the International Conference on Automated Software Engineering*. 88–99.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. 89–98.

[3] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 70–81.

[4] M. Amodio, S. Chaudhuri, and T. Reps. 2017. Neural attribute machines for program generation. *ArXiv e-prints* (May 2017). arxiv:cs.AI/1705.09231

[5] Cyrille Artho. 2011. Iterative delta debugging. *Int. J. Softw. Tools Technol. Transf.* 13, 3 (2011), 223–246.

[6] S. M. Austin, D. R. Wilkins, and B. A. Wichmann. 1991. An Ada program test generator. In *Proceedings of the Conference on TRI-Ada '91: Today's Accomplishments; Tomorrow's Expectations (TRI-Ada'91)*. 320–325.

[7] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction*. 82–92.

[8] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th Conference on Programming Language Design and Implementation*, Vol. 52. 95–110.

[9] F. Bazzichi and I. Spadafora. 1982. An automatic generator for compiler testing. *IEEE Trans. Softw. Eng.* 8, 4 (1982), 343–353.

[10] Clément Béra, Eliot Miranda, Marcus Denker, and Stéphane Ducasse. 2016. Practical validation of bytecode to bytecode JIT compiler dynamic deoptimization. *J. Obj. Technol.* 15, 2 (2016), 1:1–26.

[11] Daniel M. Berry. 1983. A new methodology for generating test cases for a programming language compiler. *ACM SIGPLAN Not.* 18, 2 (1983), 46–56.

[12] D. L. Bird and C. U. Munoz. 1983. Automatic generation of random self-checking test cases. *IBM Syst. J.* 22, 3 (1983), 229–245.

[13] Abdulazeez S. Boujarwah and Kassem Saleh. 1997. Compiler test case generation methods: A survey and assessment. *Inf. Softw. Technol.* 39, 9 (1997), 617–625.

[14] Abdulazeez S. Boujarwah, Kassem Saleh, and Jehad Al-Dallal. 1999. Testing syntax and semantic coverage of Java language compilers. *Inf. Softw. Technol.* 41, 1 (1999), 15–28.

[15] Colin J. Burgess. 1994. The automated generation of test cases for compilers. *Softw. Test. Verif. Rel.* 4, 2 (1994), 81–99.

[16] Colin J. Burgess and M. Saidi. 1996. The automatic generation of test cases for optimizing Fortran compilers. *Inf. Softw. Technol.* 38, 2 (1996), 111–119.

[17] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing compilers: A test suite and results. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 98–105.

[18] Jacqueline M. Caron and Peter A. Darnell. 1990. Bugfind: A tool for debugging optimizing compilers. *ACM SIGPLAN Not.* 25, 1 (1990), 17–22.

[19] Heung Seok Chae, Gyun Woo, Tae Yeon Kim, Jung Ho Bae, and Won Young Kim. 2011. An automated approach to reducing test suites for testing retargeted C compilers for embedded systems. *J. Syst. Softw.* 84, 12 (2011), 2053–2064.

[20] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. 2005. Type-based verification of assembly language for compiler debugging. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. 91–102.

[21] Junjie Chen. 2018. Learning to accelerate compiler testing. In *Proceedings of the 40th International Conference on Software Engineering*. 472–475.

[22] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering*. 700–711.

[23] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 266–277.

[24] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. 2017. How do assertions impact coverage-based test-suite reduction? In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 418–423.

[25] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.

[26] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*. 180–190.

[27] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *Proceedings of the 34th International Conference on Automated Software Engineering*. to appear.

[28] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2018. Coverage prediction for accelerating compiler testing. *Trans. Softw. Eng.* (2018). to appear.

[29] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

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

[31] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[32] Wai-Mee Ching and Alex Katz. 1993. The testing of an APL compiler. In *Proceedings of the International Conference on APL*. 55–62.

[33] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops*. 184–191.

[34] C. J. Burgess. 1986. Towards the automatic generation of executable programs to test a Pascal compiler. (1986).

[35] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 185–194.

[36] Maulik A. Dave. 2003. Compiler verification: A bibliography. *ACM SIGSOFT Softw. Eng. Notes* 28, 6 (2003), 2–2.

[37] Chromium developers. 2019. Chromium Testsuite. Retrieved from: https://www.chromium.org/chromium-os/testing/test-suites.

[38] GCC developers. 2019. GCC Testsuites. Retrieved from: https://gcc.gnu.org/onlinedocs/gccint/Testsuites.html#Testsuites.

[39] LLVM developers. 2019. LLVM Testing Infrastructure Guide. Retrieved from: https://llvm.org/docs/TestingGuide.html.

[40] OpenJDK developers. 2019. OpenJDK Testsuite. Retrieved from: http://openjdk.java.net/jtreg.

[41] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP (T). In *Proceedings of the 30th International Conference on Automated Software Engineering*. 482–493.

[42] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. In *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 93:1–93:29.

[43] Alastair F. Donaldson and Andrei Lascu. 2016. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing*. 44–47.

[44] Jack Dongarra, Mark Furtney, Steve Reinhardt, and Jerry Russell. 1991. Parallel loops—A test suite for parallelizing compilers: Description and example results. *Parallel Comput.* 17, 10 (1991), 1247–1255.

[45] A. G. Duncan and J. S. Hutchison. 1981. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering*. 170–178.

[46] ECMAScript. 2019. Test262: ECMAScript Test Suite. Retrieved from: https://github.com/tc39/test262.

[47] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT'08)*. 255–264.

[48] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai, and Xavier Thirioux. 2014. Testing-based compiler validation for synchronous languages. In *Proceedings of the NASA Formal Methods Symposium*. 246–251.
[49] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: Delta debugging, even without bugs. *Softw. Test. Verif. Rel.* 26, 1 (2016), 40–68.
[50] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Chen Yang, and John Regehr. 2013. Help, help, I'm being suppressed! The significance of suppressors in software testing. In *Proceedings of the International Symposium on Software Reliability Engineering*. 390–399.
[51] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *Proceedings of the International Symposium on Software Testing and Analysis*. 78–88.
[52] Kenneth V. Hanford. 1970. Automatic generation of test cases. *IBM Syst. J.* 9, 4 (1970), 242–257.
[53] Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ Trans. Syst. LSI Des. Methodol.* 9 (2016), 21–29.
[54] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will you still compile me tomorrow? Static cross-version compiler validation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. 191–201.
[55] K. Scott Hemmert, Justin L. Tripp, Brad L. Hutchings, and Preston A. Jackson. 2003. Source level debugger for the sea cucumber synthesizing compiler. In *Proceedings of the 11th IEEE Symposium on Field-programmable Custom Computing Machines*. 228–237.
[56] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 861–871.
[57] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST'16)*. 31–37.
[58] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*. 445–458.
[59] Josie Holmes and Alex Groce. 2018. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *Proceedings of the 29th International Symposium on Software Reliability Engineering*. 166–177.
[60] Josie Holmes, Alex Groce, and Mohammad Amin Alipour. 2016. Mitigating (and exploiting) test reduction slippage. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 66–69.
[61] Bernard Houssais. 1977. Verification of an Algol 68 implementation. In *Proceedings of the Strathclyde ALGOL 68 Conference*. 117–128.
[62] Plum Hall Inc. 2019. The Plum Hall Validation Suite. Retrieved from: http://www.plumhall.com/stec1.html.
[63] Bo Jiang and W. K. Chan. 2015. Input-based adaptive randomized test case prioritization: A local beam search approach. *J. Syst. Softw.* 105, C (2015), 91–106.
[64] Alexey Kalinov, Alexander Kossatchev, Alexandre Petrenko, Mikhail Posypkin, and Vladimir Shishkov. 2003. Coverage-driven automated compiler test suite generation. *Electron. Notes Theoret. Comput. Sci.* 82, 3 (2003), 500–514.
[65] Alexey Kalinov, Alexandre Kossatchev, Alexander Petrenko, Mikhail Posypkin, and Vladimir Shishkov. 2003. Using ASM specifications for compiler testing. In *Proceedings of the Abstract State Machines 10th International Conference on Advances in Theory and Practice*. 415–415.
[66] Kota Kitaura and Nagisa Ishiura. 2018. Random testing of compilers' performance based on mixed static and dynamic code comparison. In *Proceedings of the 9th International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 38–44.
[67] Donald E. Knuth. 1968. Semantics of context-free languages. *Math. Syst. Theor.* 2, 2 (1968), 127–145.
[68] Alexander S. Kossatchev and M. A. Posypkin. 2005. Survey of compiler testing methods. *Prog. Comput. Softw.* 31, 1 (2005), 10–19.
[69] Cornelis H. A. Koster. 1991. Affix grammars for programming languages. In *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*. 358–373.
[70] Philipp W. Kutter and Alfonso Pierantonio. 1997. Montages specifications of realistic programming languages. *J. Univ. Comput. Sci.* 3, 5 (1997), 416–442.
[71] Ralf Lämmel. 2001. Grammar testing. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*. 201–216.
[72] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 216–226.
[73] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 386–399.
[74] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized stress-testing of link-time optimizers. In *Proceedings of the International Symposium on Software Testing and Analysis*. 327–337.

[75] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed differential testing of interactive debuggers. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 610–620.
[76] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.
[77] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th Conference on Programming Language Design and Implementation*. 65–76.
[78] Christian Lindig. 2005. Find a compiler bug in 5 minutes. In *Proceedings of the ACM International Symposium on Automated Analysis-Driven Debugging*. 3–12.
[79] Christian Lindig. 2005. Random testing of C calling conventions. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*. 3–12.
[80] Robert Mandl. 1985. Orthogonal Latin squares: An application of experiment design to compiler testing. *Commun. ACM* 28, 10 (1985), 1054–1058.
[81] William M McKeeman. 1998. Differential testing for software. *Dig. Tech. J.* 10, 1 (1998), 100–107.
[82] L. G. L. T. Meertens and C. H. A. Koster. 1962. Basic English, a generative grammar for a part of English. In *Euratom Seminar\Machine en Talen*, Amsterdam.
[83] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *Proc. ACM Prog. Lang.* 1 (2017), 15:1–15:23.
[84] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*. 142–151.
[85] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 187–196.
[86] Tipp Moseley, Dirk Grunwald, and Ramesh Peri. 2009. OptiScope: Performance accountability for optimizing compilers. In *Proceedings of the 7th International Symposium on Code Generation and Optimization*. 254–264.
[87] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies*. 48–53.
[88] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Trans. Syst. LSI Des. Methodol.* 7 (2014), 91–100.
[89] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS'16)*. 676–679.
[90] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: Differential testing for embedded compilers. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala*. 21–30.
[91] Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. 2006. Replay compilation: Improving debuggability of a just-in-time compiler. *ACM SIGPLAN Not.* 41, 10 (2006), 241–252.
[92] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 91–97.
[93] Jibesh Patra and Michael Pradel. 2016. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*. Technical Report. TU Darmstadt, Department of Computer Science.
[94] Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. 2016. Automatic test case reduction for OpenCL. In *Proceedings of the 4th International Workshop on OpenCL*. 1:1–1:12.
[95] James Price and Simon McIntosh-Smith. 2015. Oclgrind: An extensible OpenCL device simulator. In *Proceedings of the 3rd International Workshop on OpenCL*. 1–7.
[96] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numer. Math.* 12, 3 (1972), 366–375.
[97] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation*, Vol. 47. 335–346.
[98] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*. 179–188.
[99] Hanan Samet. 1976. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 Annual Conference (ACM'76)*. 492–497.
[100] H. Samet. 1977. A machine description facility for compiler testing. *IEEE Trans. Softw. Eng.* 3, 5 (1977), 343–351.
[101] Hanan Samet. 1977. A normal form for compiler testing. In *ACM SIGART Bull.* Vol. 12. 155–162.
[102] Hanan Samet. 1977. Toward automatic debugging of compilers. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence—Volume 1*. Morgan Kaufmann Publishers Inc., 379–379.
[103] Sriram Sankar. 1989. A note on the detection of an Ada compiler bug while debugging an Anna program. *ACM SIGPLAN Not.* 24, 6 (1989), 23–31.

[104] Masataka Sassa and Daijiro Sudosa. 2006. Experience in testing compiler optimizers using comparison checking. In *Software Engineering Research and Practice*. CSREA Press, 837–843.
[105] Bauer Scotty, Pascal Cuoq, and John Regehr. 2015. Deniable backdoors using compiler bugs. *Int. J. PoC||GTFO* 0x08 (2015), 7–9.
[106] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 17–30.
[107] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Softw.: Pract. Exper.* 37, 14 (2007), 1475–1488.
[108] Emin Gün Sirer and Brian N. Bershad. 1999. Using production grammars in software testing. In *Proceedings of the 2nd Conference on Domain-specific Languages*. 1–13.
[109] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proceedings of the 13th International Symposium on Code Generation and Optimization*. 46–55.
[110] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering*. 203–213.
[111] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 849–863.
[112] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 294–305.
[113] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371.
[114] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the Asia Pacific Software Engineering Conference*. 270–279.
[115] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *Emp. Softw. Eng.* 19, 1 (2014), 182–212.
[116] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th International Symposium on Software Testing and Analysis*. 273–283.
[117] Michael Tonndorf. 1998. Ten years of tool-based Ada compiler validations an experience report. In *Proceedings of the International Conference on Reliable Software Technologies*. 176–187.
[118] A. van Wijngaarden. 1965. *Orthogonal Design and Description of a Formal Language*. Stichting Mathematisch Centrum.
[119] David B. Whalley. 1994. Automatic isolation of compiler errors. *ACM Trans. Prog. Lang. Syst.* 16, 5 (1994), 1648–1659.
[120] Klaus-Hendrik Wolf and Mike Klimek. 2016. A conformance test suite for Arden syntax compilers and interpreters. *Stud. Health Technol. Inf.* 228 (2016), 379–383.
[121] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *Trans. Softw. Eng.* 42, 8 (2016), 707–740.
[122] Gyun Woo, Heung Seok Chae, and Hanil Jang. 2007. An intermediate representation approach to reducing test suites for retargeted compilers. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*. 100–113.
[123] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 283–294.
[124] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection, and prioritization: A survey. *Softw. Test. Verif. Rel.* 22, 2 (2012), 67–120.
[125] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random program generator for Java JIT compiler test system. In *Proceedings of the 3rd International Conference on Quality Software*. 20–23.
[126] Sergey Zelenov and Sophia Zelenova. 2007. Model-based testing of optimizing compilers. In *Proceedings of the International Workshop on Formal Approaches to Software Testing and International Conference on Testing of Communicating Systems. Lecture Notes in Computer Science* 4581 (2007), 365–377.
[127] S. V. Zelenov, S. A. Zelenova, A. S. Kossatchev, and A. K. Petrenko. 2003. Test generation for compilers and other formal text processors. *Prog. Comput. Softw.* 29, 2 (2003), 104–111.
[128] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. 253–267.
[129] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28, 2 (2002), 183–200.
[130] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the International Conference on Software Engineering*. 192–201.

[131] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th Conference on Programming Language Design and Implementation*. 347–361.

[132] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. 2009. Automated test program generation for an industrial optimizing compiler. In *Proceedings of the ICSE Workshop on Automation of Software Test*. 36–43.

Received August 2018; revised May 2019; accepted September 2019