Enriching Compiler Testing with Real Program from Bug Report

Hao Zhong
Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

ABSTRACT

Differential testing is widely used to detect bugs in compilers. Its basic idea is to compile test programs with different compilers, and compare their compilation results to detect bugs. In this research line, researchers have proposed various approaches to generate test programs. The state-of-the-art approaches can be roughly divided into random-based and mutation-based approaches: random-based approaches generate random programs and mutation-based approaches mutate programs to generate more test programs. Both lines of approaches mainly generate random code, but it is more beneficial to use real programs, since it is easier to learn the impacts of compiler bugs and it becomes reasonable to use both valid and invalid code. However, most real programs from code repositories are ineffective to trigger compiler bugs, partially because they are compiled before they are submitted.

In this experience paper, we apply two techniques such as differential testing and code snippet extraction to the specific research domain of compiler testing. Based on our observations on the practice of testing compilers, we identify bug reports of compilers as a new source for compiler testing. To illustrate the benefits of the new source, we implement a tool, called LeRe, that extracts test programs from bug reports and uses differential testing to detect compiler bugs with extracted programs. After we enriched the test programs, we have found 156 unique bugs in the latest versions of `gcc` and `clang`. Among them, 103 bugs are confirmed as valid, and 9 bugs are already fixed. Our found bugs contain 59 accept-invalid bugs and 33 reject-valid bugs. In these bugs, compilers wrongly accept invalid programs or reject valid programs. The new source enables us detecting `accept-invalid` and `reject-valid` bugs that were usually missed by the prior approaches. The prior approaches seldom report the two types of bugs. Besides our found bugs, we also present our analysis on our invalid bug reports. The results are useful for programmers, when they are switching from one compiler to another, and can provide insights, when researchers apply differential testing to detect bugs in more types of software.

ACM Reference Format:
Hao Zhong. 2022. Enriching Compiler Testing with Real Program from Bug Report. In *ASE 2022.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.475/123_4

ASE 2022, 10-14 October, 2022, Ann Arbor, Michigan, United States

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

As compilers are among the most popular and important software and compiler bugs can lead to disastrous consequences, detecting their bugs has been a hot research topic [34]. If multiple implementations shall follow a specification, differential testing [65] presents a practical test oracle to detect bugs: given the same inputs, these implementations shall produce identical outputs. A programming language typically has more than one compiler, and it shall follow specific standards. For example, the International Organization for Standardization defines the standard for C++ [17], and all C++ compilers shall follow this standard to compile code. As programming languages have specifications and multiple compilers, researchers [34] have used differential testing to detect bugs in compilers. In differential testing, the diversity and quality of test inputs determine the number and importance of detected bugs. Chen *et al.* [41] reviewed 85 papers, and they found that 51% papers work on the generation of test programs.

Although commercial test suites such as PlumHall [20] and SuperTest [21] are available, detecting compiler bugs needs more test programs. Researchers generate test programs through two sources:

1. Random-based approaches generate syntactically valid programs, and to generate valid programs, random-based approaches often take the grammars of programming languages as their inputs. Early approaches are traced back to 1970s [53, 70]. As a more recent tool, Yang *et al.* [85] propose CSmith that is based on random test generation and generates C programs. Nagai *et al.* [66] generate random C arithmetic expressions to detect arithmetic-optimization bugs. Zhang *et al.* [88] propose an approach to generate effective programs by identifying those equivalent ones. Chen *et al.* [42] propose an approach that tunes the configurations of CSmith to generate more diverse programs.

2. Mutation-based approaches change given programs to generate more test programs. Sun *et al.* [73] mutate variable and function names to generate programs. Holler *et al.* [54] mutate programs by traversing their syntax trees. Le *et al.* [58, 59] propose a mutation technique called EMI. Given a C program, EMI compiles the code, and executes the compiled code to collect its executed source lines. EMI then removes unexecuted source lines, and compiles the remaining code again. By comparing the outputs of the code after the removal with those of the original code, researchers (*e.g.*, [58, 72]) find that optimization bugs in compilers cause different outputs. EMI has been extended to detect bugs in OpenCL [63] and Simulink [46] compilers. Even if the mutation seeds are real code, mutated programs are not real, and programmers may not write such programs in practice.

Compared with random code, it is more beneficial to use real code, since it is easier to learn the importance of a bug and it becomes reasonable to use real invalid programs as meaningful inputs for compiler testing (see the listed benefits in Section 2.2 for

details). In this paper, we identify bug reports as a rich source to collect such programs, based on the following observations:

1. Bug reports of compilers contain many useful test programs that are derived from real projects. Although researchers can submit their random or mutated programs, most bug reports of compilers are submitted from real users, and their programs are derived from real projects. To analyze faulty locations, those programs are often reduced. Programs from bug reports are often challenging, and it is unlikely to generate such programs via randomness or mutations. Although researchers can submit random programs, most programs in bug reports come from real development, in that most bug reporters are real users of compilers.

2. From bug reports, it is feasible to extract real test programs for regression testing. As test programs in bug reports are useful, we notice that the test program of at least a bug report [2] has been added to the test suites of compilers [3]. As programmers add only few test programs from bug reports to their test suites, most test programs from bug report are still useful for regression testing. Indeed, we have found recurring bugs, when we use the test programs from a compiler to test this compiler (see long-standing bugs and controversial programs in Section 4.2).

3. The test programs of a compiler are useful to test other compilers. As described in the guideline of `clang` [1], its test suite contains some test programs from the test suite of `gcc`. This practice confirms that the test programs of a compiler are useful for other compilers. When users encounter a compiler bug, users often report test programs to their used compiler. As a result, even after a bug is fixed in a compiler, other compilers can still leave similar bugs unfixed.

Based on our observations, we conduct the first experience study, in which we applied two automated software engineering techniques in compiler testing. Our contributions are as follows:

- *The first exploration on an effective way of using real test programs in compiler testing.* We are the first to advocate the using of real programs in compiler testing. An arbitrary piece of real code is ineffective to detect compiler bugs. To resolve this problem, we identify bug reports as a source to mine effective real test programs for compiler testing. To illustrate the benefits of the new source, we apply the code extraction [26, 55, 69, 90] and differential testing [34] techniques, and implement the two techniques in a tool called LeRe to support our study.
- *An experienced report that provides positive empirical evidences of our new source.* We conducted an experience study on `gcc` and `clang`. In total, we have found 156 bugs. Among them, 103 bugs are confirmed as valid by their developers, and 9 bugs are already fixed after we reported them. Our found bugs include 59 accept-invalid bugs and 33 reject-valid bugs. The prior approaches [58, 59, 85] do not report the two types of bugs. Meanwhile, we found new crashes and wrong-code bugs as they did.
- *Lessons on false alarms of differential testing.* We report the limitations of differential testing, in its application of compiler testing. Our results show that although the differences between compilers are typically bugs, there are cases where differences are not considered as bugs. Most of such cases indicate undefined behaviors or conflicts in the C++ standard.

2 RESEARCH METHODOLOGY

According to its website [4], ASE2022 calls for both technical research papers and experience papers. An experience paper describes a significant experience in applying automated software engineering technology, and its evaluation criteria include the importance of the problem, the insights from the study, and the identified lessons. In this experience paper, we applied a software analysis technique and a testing technique to enrich the test inputs for compiler testing. For the techniques, Section 2.1 introduces their background, and Section 3.1 introduces the implementation. For the problem, Section 1 introduces why it is desirable to extract test programs from bug reports, and Section 2.2 further analyzes its importance. For the insights, Section 4 shows that test programs from bug reports detected more than a hundred compiler bugs. For the lessons, Section 5 shows that differential testing has inherent limitations. Our research methodology is customized for experience papers, and we do not list explicit research questions like empirical studies.

2.1 Automated SE Technology

This experience paper leverages a software analysis technique and a software testing technique:

1. The software analysis technique. As a software analysis technique, extracting code snippets from a software engineering document is intensively studied. Bacchelli *et al.* [27, 28] propose approaches that extract code snippets from emails, and further extend their approaches to extract more types of contents [25, 26]. Besides emails, researchers have proposed approaches to extract code snippets from StackOverflow [69] and research articles [35]. In this experience study, we extend this technique to extract programs from the bug reports of `gcc` and `clang`. As an underlying technology, the code-snippet extraction has been an integration of various approaches such as linking API and its learning resources [47], detecting API documentation errors [90], and learning patterns of self-admitted technical debts [87]. As extracting code snippets from SE documents is intensively studied, it shall be feasible to extract test programs from bug reports.

2. The testing technique. Differential testing is a widely used test oracle to determine the correctness of test outputs. To compete the market shares, there are typically multiple projects that implement similar or even identical functions. The basic idea of differential testing lies in that given the same inputs, the implementations shall produce the same outputs. As a test oracle, differential testing has been introduced to detect bugs in refactoring [48], cross-language APIs [91], web browsers [45], virtual machines [64], crashes [56], JVMs [89], SSL/TLS [44], and compilers [85]. When differential testing is introduced to test compilers, the oracle is that compilers shall consistently accept and reject code.

In this experience paper, we implemented a tool called, LeRe, that extracts programs from bug reports, and compiles extracted programs with differential testing.

2.2 Importance of Target Problem

Real programs may not effectively trigger compiler bugs, since they are compiled before they are submitted to code repositories. Even the mutated code of real programs is ineffective to detect compiler bugs. For example, Le *et al.* [58, 59] mutated both random and real

Gerhard Steinmetz **2017-02-07 17:19:41 UTC**

```
Other test cases :

$ cat z1.c
void f()
{
  void g()
    void a[( {void b} )];
}
```

(a) Program

Eric Gallager **2017-07-31 02:54:05 UTC**

```
Confirmed that gcc still ICEs, although I
not... I'll leave the "ice-on-valid-code"
```

(b) Comment

```
$ gcc-7-20170205 -c pr30552.c
pr30552.c: In function 'fun':
pr30552.c:6:5: internal compiler error: Segmentation fault
     int a[({void h(){}2;})];
     ^~~
0xbf633f crash_signal
        ../../gcc/toplev.c:333
```

(c) Error message

Figure 1: Three major elements in bug reports

code to generate test programs, but through mutated real code, they found only one gcc bug. Based on our observations in Section 1, we envisage that it is potential to extract test inputs from bug reports. Towards this direction, our target problem is as follows:

> How to extract test programs from bug reports, and how effective is the new source to detect compiler bugs?

For the first question, Section 3 shows that our support tool is accurate to extract test programs from bug reports. For the second question, Section 4 shows that test programs from bug reports are effective to detect compiler bugs. Indeed, we found more than a hundred compiler bugs, and several types of bugs (*e.g.*, `accept-valid` bugs) that are rarely reported by the prior approaches (see Section 4.5.1 for details). Our positive results can unleash the benefits:

Benefit 1. It is feasible to learn the impacts of bugs if test programs are real code. It is feasible to contact the authors of real programs and to learn the impacts of a bug. For example, the test program of a bug report [12] comes from a real project, OpenMandriva. In this bug report, a developer mentioned that OpenMandriva switched its compiler from `gcc` to `clang`, partially due to this bug. Indeed, when we reported our found bugs, some compiler developers asked whether our programs were real code, so that they can determine the severity of our reported bugs.

Benefit 2. It becomes reasonable to use invalid code, when test programs are real code. When generating test programs, most approaches (*e.g.*, Yang *et al.* [85]) require that generated programs are valid code, but several approaches (*e.g.*, Holler *et al.* [54]) generate test programs from seeds that triggered invalid behaviors. Although it is straightforward to generate invalid programs, it is difficult to generate meaningful invalid programs. If they are randomly generated, invalid programs are often meaningless, since programmers may not write such invalid programs. As a result, all the prior approaches filter their generated invalid code. In contrast, if they are real code, invalid programs are meaningful, since programmers in real development have written such code. As most programs on bug reports are real code, it is no longer necessary to identify and filter invalid code, if test programs are extracted from bug reports.

Table 1: Extracted features

Category	ID	Feature
structure	F_1	the number of compilation errors
	F_2	the number of grammar errors
word	F_3	the number of keywords
	F_4	the number of compiler commands
punctuation	F_5	the number of splashes
	F_6	the number of colons
	F_7	the number of dollars
	F_8	the number of semicolons

Besides C/C++, for other popular languages, there is typically at least a popular compilers, and our general idea is beneficial for all popular programming languages. Even if a language is new, it is beneficial to keep an eye on the progress of competitors. The bug reports of a compiler are beneficial for the testing of other compilers and the regression testing for this compiler. The benefits of bug reports are not once for all, since even mature compilers (*e.g.*, `gcc` and `clang`) receive hundreds of new bug reports each day.

3 SUPPORT TOOL

Section 3.1 introduces the implementation. Section 3.2 introduces its highlight. Section 3.3 presents its effectiveness.

3.1 LeRe

Given a bug report, LeRe extracts programs from its attachments (Section 3.1.1) and descriptions (Section 3.1.2), and it compares the compilation results to detect bugs (Section 3.1.3).

3.1.1 Parsing Attachment. Bug report can have attachments in various formats. Some attached files are binary files, but the current implementation of LeRe ignores binary files. We notice that such binary files are often compiled code, intermediate code, and compressed files of such files. It is infeasible to use these files as test programs, but they can be useful to diagnose compiler bugs.

If a file is a textual file, LeRe first identifies programs through their file names. In particular, if a file name contains the word, *testcase*, LeRe considers it as a program. In contrast, if a file name contains words such as *patch*, *fix*, *diff*, *script*, *log*, and *trace*, it considers it as not a program.

If a file name does not have the above hints to determine whether it is a program, LeRe uses the Microsoft C++ compiler [22] to parse it and to determine whether it is a program. As some files are invalid code, they can contain compilation errors. To extract such files, if a file has fewer than ten compilation errors, LeRe considers it as a program. Here, instead of `clang` or `gcc`, we select another compiler, since both `clang` and `gcc` can wrongly reject valid programs (see Table 4 for such examples). Selecting either of the two compilers can introduce bias. For example, if we select `gcc`, we will lose all `reject-valid` bugs of `gcc`. To resolve this issue, we select the Microsoft C++ compiler.

3.1.2 Analyzing Description and Comment. Different from traditional natural language corpus (*e.g.*, newspapers), as shown in Figure 1, a bug report is a mixture of several elements such as programs (Figure 1a), comments (Figure 1b), and error messages (Figure 1c). LeRe needs to extract programs from bug reports of compilers.

Marc Glisse **2013-11-24 23:11:57 UTC** Description [reply] [−]

```
enum class E : int { prio = 666 };
void f (int) __attribute__((constructor(E::prio)));
```

is accepted with -std=c++11 whereas the conversion should require an explicit cast. default_conversion -> decay_conversion -> decl_constant_value_safe returns: <integer_cst 0x7ffff662efc0 type <enumeral_type 0x7ffff6624b28 E> constant 666> and we then accept any integer_cst.

The example comes from ~~PR 59211~~.

Figure 2: The bug report of gcc59281

The programs of bug reports are more difficult to be extracted than those of the other documents. First, the programs of bug reports are much shorter. For example, in Figure 5, three out of the six programs have fewer than two lines of code. When programs are too short, it becomes more difficult to distinguish them from other elements. Second, a program can be invalid. As such programs can have compilation errors, the borderline between programs and other elements is further weakened.

To handle the problem, we extract three different types of features, and Table 1 shows our extracted features.

1. Structure features (F_1 and F_2). For structure features, Bacchelli *et al.* [26] use an island parser to extract code features, and Zhong and Su [90] use an NLP parser to extract NLP features. LeRe combines both strategies. For each paragraph, it uses the Microsoft C++ compiler to collect the number of compilation errors (F_1), and uses the language tool [18] to collect the number of grammar errors (F_2).

2. Word features (F_3 and F_4). For each paragraph, F_3 counts the number of keywords, and F_4 counts the number of parameters.

3. Punctuation features (F_4 to F_8). The prior tools [26, 90] use punctuation features to extract programs in Java. For C++, LeRe uses different punctuation features: for each paragraph, F_4 to F_8 count the numbers of slashes, colons, dollars, and semicolons (*i.e.*, "/", ":", "$", and ";").

To extract programs, LeRe is built on the decision tree learning [50]. Decision tree is a supervised classification technique. It constructs an if-else tree to classify instances. Each internal node denotes a variable, and each leaf denotes a class. Furthermore, LeRe uses AdaBoost [51] to improve its effectiveness. AdaBoost is a meta-level learning technique. It combines outputs of weak classifiers into a weighted sum to predict better outputs.

LeRe predicts three types of elements in bug reports.

1. Programs. As shown in Figure 1a, some bug reports provide C++ programs. LeRe needs these programs to enrich the test inputs of testing compilers.

2. Comments. As shown in Figure 1b, comments are written in natural languages (English in particular).

3. Others. LeRe predicts other elements than programs and comments into this category (*e.g.*, the error message in Figure 1c).

Several paragraphs can construct a single program, but as the prior approaches do, LeRe identifies programs by paragraphs. To handle this problem, LeRe merges programs into one, if there are no other elements between them. Meanwhile, a paragraph can have multiple types of elements. For example, Figure 1a shows that a program can have a corresponding command line (`$cat z1.c` in this example) or an introduction sentence (*e.g.*, "the program is as follows:"). If we include them in a program, both compilers will fail to compile it. To handle the problem, LeRe implements a set of

```
error: constructor priorities must be integers from 0
to 65535 inclusive
void f (int) __attribute__((constructor(E::prio)));
```

Figure 3: The error message of gcc, after gcc 59281 is fixed.

heuristics to remove command lines and introduction sentences. In particular, if a paragraph is identified as a program, LeRe checks whether the first line starts with "$" or "#". If it is, LeRe removes it, since it is a command line. In addition, LeRe checks whether the first line ends with ":". If it is, LeRe removes it, since it is an introduction sentence.

3.1.3 Execution and Comparison. LeRe extracted programs from collected bug reports, and compiled them with `gcc` and `clang`. For each test program, it initiates a thread, and compiles the program with `-c` to avoid link errors. All the other settings are default, but it shall be feasible to enumerate more compiler options, and more compilers bugs (*e.g.*, optimization bugs) can be thus detected. As messages from compilers are automatically generated, they follow strict formats. After a thread terminates, it parses its return messages to determine whether a compiler accepts or rejects a test program. The results fall into five categories:

(1) *The AA category.* Both compilers accept the program.
(2) *The RA category.* `gcc` rejects, but `clang` accepts.
(3) *The AR category.* `gcc` accepts, but `clang` rejects.
(4) *The RRI category.* Both compilers reject the program, and the error lines are identical.
(5) *The RRD category.* Both compilers reject the program, but the error lines are different.

We found that the majority of programs fall into the *AA* and *RRD* categories, and only a small portion of programs fall into the *RA* and *AR* categories. In our study, we analyze and report programs of only the *RA* and *AR* categories. It is feasible to detect more bugs from the other categories. For example, Le *et al.* [59] show that even if a program compiles under different optimization flags, they can produce different outputs given the same inputs, so the *AA* category can still detect `wrong-code` bugs. As another example, although two compilers both reject a program, the compilation errors and their locations can be different, so the *RRD* category can also detect bugs. In our study, we did not analyze such cases, since it requires test inputs for compiled code, and it needs much more human efforts to determine whether such differences are bugs.

3.2 Highlight

In this section, we introduce the highlights of LeRe. As shown in Figure 2, the bug reporter, Marc Glisse, pointed out that the second line of the program requires an explicit cast, but `gcc4.9.0` wrongly accepts the program. The developers of `gcc` fixed the bug, and the fixed `gcc` rejects the program, with the error message as shown in Figure 3. LeRe extracted the program, and compiled it with the latest `gcc` and `clang` (*i.e.*, `gcc9.0.0` and `clang7.0.0`). It found that although the latest `gcc` rejects the code, the latest `clang` still accepts it. We reported the inconsistency to `clang`. The bug was fixed [6], one day after we reported it.

As shown in Figure 2. The `gcc` bug report does not attach the program, but embeds the program in its description. If we feed descriptions to compilers, we will not find the `clang` bug [6], since both

Table 2: The precision, recall and f-score of different features.

	All			F_1			F_2			F_3, F_4			F_5-F_8		
program	0.977	0.968	0.973	0.414	0.643	0.504	0.589	0.777	0.67	0.527	0.689	0.597	0.51	0.884	0.647
comment	0.944	0.964	0.954	0.553	0.469	0.508	0.617	0.647	0.632	0.514	0.704	0.594	0.588	0.294	0.392
other	0.978	0.966	0.972	0.464	0.277	0.347	0.512	0.324	0.397	0.741	0.239	0.362	0.872	0.668	0.756
average	0.966	0.966	0.966	0.477	0.463	0.453	0.573	0.583	0.566	0.594	0.544	0.518	0.656	0.615	0.598

compilers will reject the description. LeRe extracts the program, with its trained classifier (Section 3.1.2).

As shown in this example, our solution has the two benefits as described in Section 2. For the first benefit, as we extract the program from a bug report, the program in Figure 2 is a real program that is simplified from a real project. For the second benefit, our reported program is an invalid program. The prior approaches are unlikely to detect this bug, since they generate only valid programs. As a result, we have complemented the other two sources such as random programs and mutated programs. First, our bug report shows that the compiler can wrongly accept invalid programs. As the code is invalid, the compiled code is also invalid and its behavior is against the intension of programmers. When the bug occurs, the compiler does not report any messages, so it is difficult for programmers to manually identify it. However, the prior approaches cannot detect this type of bugs. Second, our program is written in C++, and its bug is located in the C++ component. In summary, LeRe has the following two highlights:

Highlight 1. It extracts real programs, and its extracted programs include both valid and invalid ones. Yang *et al.* [85] randomly generate programs as test inputs of compilers. Le *et al.* [58, 59] mutate programs to generate more test inputs for compilers. Although they have detected hundreds of bugs, both approaches can generate programs that human programmers will never write, and such programs can trigger superficial bugs. In addition, the prior tools are unlikely to generate some types of programs. For example, the prior tools generate valid code, but programmers can write invalid code that is wrongly accepted by compilers. With LeRe, we reported bugs that are triggered by invalid programs, and some of such reports are already confirmed and fixed (see Section 4.5.1).

Highlight 2. It is able to detect bugs in C++. Although it is difficult for a tool to generate C++ programs, Sun *et al.* [73] show that users have filed many bug reports in C++. From such bug reports, LeRe has the potential to detect more types of bugs in more languages (*e.g.*, accepting invalid programs).

Besides the above achievements, we notice that LeRe has additional benefits. For example, our extracted programs of LeRe are reduced, since in bug reports, developers often reduce programs to locate which lines introduce bugs. As shown in Figure 2, our program has only two lines of code. As another example, we have found bugs that are unknown to both compilers. As LeRe extracts programs from bug reports, it is unsurprising that our found bugs are already known in the other compilers. In this example, our reported `clang` bug is similar to the original `gcc` bug. If a program is able to detect a bug in a compiler, the program is often written in a controversial or complicated way. As a result, our extracted programs can trigger other unknown bugs (see Section 4 for details).

3.3 Effectiveness

In our application, if other elements are wrongly identified as programs, they increase the execution time, but shall not change the results, since both compilers will reject such programs. If programs are wrongly identified as other elements, we can miss compiler bugs, since these programs are not used as test inputs. As a result, recalls are more important in our application. To construct the golden standard of the evaluation, we manually classified 1,868 paragraphs. We conduct a tenfold cross validation on the dataset to evaluate the effectiveness of LeRe.

Table 2 shows the results. Column "All" lists the results with all our features, and the other columns show the results with only given features. The three subcolumns of each column list precision, recall, and f-score, and the three rows "code example", "comment", and "other" show the results for classifying code examples, comments, and others, respectively. Row "average" shows the averages of the three categories. Table 2 shows that a single type of features is insufficient to achieve high f-scores. In particular, F_1 achieves the lowest f-score, in that some programs are invalid. The results highlight the challenges of our research problem. However, Column "All" shows that the combination of all the features accurately extracts code examples from bug reports.

4 EXPERIENCE IN THE WILD

Although the results in Section 3.3 are quite positive, the purpose of our study is more than presenting the effectiveness on datasets. We next introduce our experience in the wild. Since April 2018, we have used LeRe to detect bugs in `gcc` and `clang`. At the beginning, we used `gcc 8.0.1` and `clang 7.0.0`. When we started to report our found bugs, in June 2018, a `gcc` developer complained that `gcc 8.0.1` is not the latest version anymore, and asked us to try our bugs on the latest version. We updated our `gcc` to `9.0.0`. As a result, most of our bugs are found in `gcc 9.0.0` and `clang 7.0.0`. However, after developers reproduced our bugs, they updated the versions of our bug reports to the versions of their own `gcc` or `clang`. Our found bugs are listed on our project website: https://github.com/drhaozhong/otherbugreport

In summary, we achieved the following goals:

1. Many confirmed bugs. In total, we have found 156 bugs. Among them, 103 bugs are confirmed as valid by their developers, and 7 bugs are already fixed after we reported them. As a comparison, Le *et al.* [58] found 147 bugs, 77 of which are found in the latest version. Our bugs are comparable with theirs. All our bugs are found in the latest versions of `gcc` and `clang`, and they are unknown before we submit them. It is feasible to tune approaches according to bugs, if bugs are already known. Intuitively, it is more challenging to detect unknown bugs.

2. New types of bugs. In total, we found 59 accept-invalid bugs and 33 reject-valid bugs. The prior approaches [58, 59, 85] do not

Table 3: Overall result.

	gcc	clang	total
fixed	5	4	9
new	31	18	49
moved	0	1	1
reopen	0	1	1
suspended	4	0	4
won't fix	0	3	3
	40	27	67
duplicate	24	5	29
unconfirmed	7	0	7
invalid	16	37	53

report the two types of bugs. Meanwhile, we found new crashes and wrong-code bugs as they did.

3. Several long-standing bugs. For both compilers, we found several long-standing bugs. Although these bugs were fixed in one compiler about ten years ago, they have been in the other compiler for ten years, and most of them are unknown, until we reported our found bugs. Our bug reports attracted developers' attention, and finally they are fixed in the latest versions.

4. Bugs that are never reported in any compilers. Some of our programs trigger quite different bugs from their original bug reports. Some of such bugs are never found in any compilers, before we report them.

4.1 Setup

Hardware and compilers. We have performed our testing on a machine (E5-2620v4 Xeon CPU) running Ubuntu 18.04 (x86_64). For both compilers, we used their default settings. As developers often reduce programs, most of our extracted programs are short, and do not even have the `main` functions. To avoid link errors, we add the flag, `-c`, and focus on the compiling process.

Collecting known bug reports. We used LeRe to download the bug reports of `gcc` and `clang`. Among downloaded bug reports, we analyzed only reports whose resolutions are `fixed` and are not `duplicate`. As bug reports are not duplicate, their programs are unlikely repetitive. To show the benefits of LeRe, we analyzed the bug reports that are related to the C++ components of `gcc` and `clang`. In total, we analyzed 9,708 bug reports from `gcc`, and 3,213 bug reports from `clang`.

4.2 Overall Result

With LeRe, we have obtained the following achievements:

1. LeRe found new types of bugs that are never reported by the prior approaches. In total, we have filed 156 bug reports, and most of them are bugs in the C++ components of `gcc` and `clang`. Among them, 103 bugs are confirmed as valid, and 9 bugs are already fixed. Besides 8 crashes and 1 `wrong-code bug`, LeRe found 59 `accept-invalid` bugs and 33 `reject-valid bugs`. In an `accept-invalid` bug, a compiler accepts an invalid program, and in a `reject-valid` bug, a compiler rejects a valid program. The prior approaches seldom report the two types of bugs.

2. LeRe found bugs that do not appear in prior bug reports. As LeRe reuses programs from other compilers, it is unsurprising

```
class test {
  friend int bar(int = true);
};
```

(a) The program of `gcc59480`

```
class Test{
 friend const int getInt(int inInt = 0);
};
```

(b) The program of `gcc86502`

Figure 4: Our bug report (`gcc86502`) was marked as a duplicate of a previous bug report (`gcc59480`). Although `gcc59480` was reported in 2013, it was fix, only one month after we reported our duplicate bug.

that some of our reported bugs are similar to original bug reports of the other compilers. For example, the program in Figure 6e is extracted from a `clang` bug report [10]. The two bugs are identical. When we reported the `gcc` bug, we even copied a sentence from the original `clang` bug report. However, LeRe also found bugs that have little connections with their original bug reports. For example, in Section 4.3, we introduce that our report bug (`gcc86502`) speeds up the repair process of a known bug (`gcc59480`). Our code example is extracted from a `clang` bug report [8]. `clang` crashes on this program, which is irrelevant with our reported `gcc` bug (`gcc59480`). As another example, the program in Figure 7a is extracted from a `clang` bug report [5]. `clang` miscompiles the program, but `gcc` crashes when compiling it. The program triggers a `wrong-code` bug in `clang`, but an `ice` bug in `gcc`. Although some of our reported bugs are known in other compilers, our results show that LeRe found bugs that are *never reported in any compilers.*

3. LeRe found several long-standing bugs. As shown in Section 4.3, one of our duplicate bugs raised attention to a bug that was reported in 2013, and it was finally fixed one month after we reported it. Indeed, besides this bug, we found other long-standing bugs. For example, Figure 8 shows a program. The program is extracted from a `clang` bug report [9] that was reported in 2010. As the `clang` bug says, the program comes from an earlier `gcc` bug [11] that was reported in 2007. Both the `gcc` bug and the `clang` bug were marked as fixed. However, we found that even the latest `gcc` and `clang` handle it differently, and we reported this bug to `gcc` [13]. Its developers explained the issue: "*CWG 1839 hasn't been resolved yet and doesn't even have a proposed resolution in the issues list...If 1839 is going to make more changes in this area then this PR should be suspended until any change happens*".

4. LeRe found some controversial programs. For example, in our bug report [7], we find that `gcc4.8`, `5.2`, `9.0`, and `clang3.6` accept a program, but `gcc6.0` and `clang7.0` reject it. It is difficult for other tools to generate such programs, since a machine typically does not understand whether the differences are reasonable.

5. Most reported programs are reduced. The programs in Figures 5d, 5e, 5f, and 6d have only one or two lines of code. When developers fix a compiler bug, they often manually reduce programs. As our programs are often reduced, it saves the effort to locate bugs.

4.3 Quantitative Result

Table 3 shows our overall results. In total, we find 156 bugs. Among them, 103 bugs are real bugs. In particular, 9 bugs are *fixed* after we report them; 49 bugs are confirmed as *new* bugs; 1 bug is *moved*; 1

```
template < typename > struct traits;
template < typename T > struct X{
 X & operator = (X &&) noexcept
  (traits < T >::foo ());
};
template < typename T >
X < T > &
X < T >::operator = (X &&) noexcept
(traits < T >::bar ()){
  return *this;
}
```

(a) clang accepts the code, but it is invalid, since the declaration has a different exception specifier.

```
#include <typeinfo>
#include <iostream>
template<class A,class B> void f(){
  std::cout << std::boolalpha
   << (typeid(A)==typeid(B)) << '\n';
}
int main(){
  f<void()const,void()>();
}
```

(b) clang accepts the code, but the code is invalid, since it violates [dcl.fct].

```
template<class T>
class A{
 static T a;
};
template<class T>
T A<T>::a;
class B{ };
template
int A<B>::a;
```

(c) clang accepts the code. The code is invalid for the explicit template instantiation, because the type given for the static variable does not match the one in the class template.

```
int x =
 reinterpret_cast<const int&&>(1.0f);
```

(d) gcc accepts the invalid code. It violates [expr.reinterpret.cast].

```
auto l = [=]{};
```

(e) gcc accepts the code, but the code is invalid, since it violates [expr.lambda] p9.

```
struct S { struct T {}; };
::decltype(S())::T st;
```

(f) gcc accepts the code, but the code is invalid, since it expects unqualified-id.

(a) https://bugs.llvm.org/show_bug.cgi?id=38141; (b) https://bugs.llvm.org/show_bug.cgi?id=37846; (c) https://bugs.llvm.org/show_bug.cgi?id=38205; (d) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86633; (e) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86499; and (f) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86500

Figure 5: `accept-invalid` bugs

```
template<typename... Args>
void spurious(Args... args){
    (... + args).member;
}
int main(){}
```

(a) clang rejects the code, since it fails to parse the fold expression.

```
struct X {
    template <class T>
    void foo();
};
struct Base {
    X get();
};
template <class >
struct Derived : Base{
    void foo() {
        auto result = Base::get();
        result.foo<void>();
    }
};
template struct Derived<int>;
```

(b) clang determines that the template names are dependent, but they are not.

```
#include <stdio.h>
struct X {
 X(){printf("X(): this=%p\n", this);}
 X(const X& other) {
  printf("X(const &X):this=%p,
   other=%p\n", this, &other); }
 ~X() { printf("~X(): this=%p\n",
  this); }
 operator bool() {
  printf("X::operator bool():
   this=%p\n", this); return true; }
};
int main() {
 X x = X()? : X();
}
```

(c) gcc rejects the code, since it miscalculates the type of a return value.

```
#include <stddef.h>
::nullptr_t n;
```

(d) clang rejects this code. [depr.c.headers] says that it is valid.

```
struct A {
 static int const B = sizeof B;
 };
```

(e) gcc rejects the valid code, but clang rejects it.

```
void f(char*);
int &f(...);
int &r = f("foo");
```

(f) gcc, icc, and MSVC reject the valid code, but clang accepts it.

(a) https://bugs.llvm.org/show_bug.cgi?id=38282; (b) https://bugs.llvm.org/show_bug.cgi?id=38299; (c) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86184; (d) https://bugs.llvm.org/show_bug.cgi?id=38216; (e) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86431; and (f) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86498

Figure 6: `reject-valid` bugs

bug is initially marked as invalid but later is marked as *reopen*; 4 bugs are *suspended* before assigning to programmers; and 3 bugs are marked as *won't fix*. When we reported our bugs, we selected the default severity, since Tian *et al.* [77] show that it is difficult to select proper the severity of a bug report and programmers tend to ignore the severity. Like Le *et al.* [59], in our study, more gcc bugs are confirmed, since gcc developers pay more attention to the comparability of compilers.

Although they are valid, 29 bugs were marked as *duplicate*. Bettenburg *et al.* [32] argue that duplicated bug reports are not harmful. Although our reported bugs were marked as duplicate, we found that they report different programs, and our duplicate bugs boost the repair process of some bugs. For example, in 2013, a developer, named Tobias Burnus, reported that gcc accepts the program as shown in Figure 4a, but `clang` rejects it [12]. The reporter believed that the program violates the C++ specification. OpenMandriva [19] is a Linux operation system. In the follow-up discussion of the bug report [12], another developer mentioned that OpenMandriva switched its compiler from gcc to `clang`, partially due to this bug. However, this bug was not fixed before we reported our found bug. We reported a bug on July 12th 2018 [16], and Figure 4b shows our program. Our reported bug (`gcc86502`) was resolved as a duplicate of `gcc59480`. After that, gcc developers submitted three patches on July 18th 2018, July 19th 2018, and August 7th 2018. Finally, the bug was fixed five years after it was first reported.

In total, 7 bugs are *unconfirmed*, and 53 bugs are marked as *invalid*. Section 5 presents our invalid bugs.

4.4 Bug Category

The gcc developers classified some of our bug reports with keywords. Following their definition, we classified all our bug reports. Table 4 shows our classification results: `accept-invalid` denotes

```c
#include <stdio.h>
template<typename T> static char const * f(
    T *t) {
 T u(*t);
 u.x = "hello world";
 printf("%s\n", u.x);
 return "initialized";
}
int main() {
 union { char const *x = f(this); };
 printf("%s\n", x);
}
```

(a) An internal compilation error of gcc.

```c
struct A {
 int* a;
 A(int a) : a(new int(a)) {}
 ~A() { delete a; }
 A(const A&) = delete;
 A(A&& other) { a = other.a; other.a = 0;
     };
 operator bool() { return true; }
 int operator*() { return *a; }
};
static A makeA(int x) { return A(x); }
int main() {
 A c = makeA(42) ?: makeA(-1);
 return *c;
}
```

(b) A wrong-code error.

```c
class X {
 public:
  int i;
};
inline const int& OHashKey(const X& x) {
 return x.i;
}
int main() {
 extern const int& OHashKey(const X& x);
 X x;
 return OHashKey(x);
}
```

(c) A link failure of gcc.

(a) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86182; (b) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86385; and (c) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86208

Figure 7: Other bugs

Table 4: Bug category of our detected bugs.

	gcc	clang	total
accept-invalid	40	19	59
reject-valid	22	11	33
ice	6	2	8
other	5	1	6

that a compiler wrongly accepts an invalid program; `reject-valid` denotes that a compiler wrongly rejects a valid program; and `ice` denotes internal compiler errors which often cause crashes.

As shown Table 4, we find 92 `accept-invalid` and `reject-valid` bugs. The results indicate that most of our programs are borderline cases, which introduce controversial compilation results. It is rather difficult for a tool to generate such borderline cases, especially when the differences shall be meaningful to programmers. As a result, the prior approaches [58, 85] did not find the two types of bugs.

Besides the two types of bugs, Table 4 shows 8 `ice` bugs, and 1 `wrong-code` bug. Most `ice` bugs are crashes, and most bugs of Yang *et al.* [85] are crashes. Le *et al.* [58] define that `wrong-code` refers to runtime errors of compiled code. As we did not execute compiled code, we did not report our bugs as `wrong-code` bugs, but one of our reported bugs was marked as `wrong-code` by `gcc` developers [15]. Le *et al.* [58] detected performance bugs. When LeRe compared the results of compilers, we found that both compilers hang on some programs. At that time, we did not know that they were performance bugs, and we ignored them.

LeRe detected even more types of bugs than the above-mentioned bugs. We present their programs in Section 4.5.3.

4.5 Sample Bug

As LeRe extracts programs from reported bugs, it has the potential to detect any type of bugs, if that type of bugs were once reported. However, as the first work in this research line, we did not prepare the required inputs of some bugs. For example, as we did not prepare test inputs for compiled code, it is less effective to detect `wrong-code` bugs than the prior work [58, 85]. As another example, we have not detected any optimization bugs, since we did not try corresponding flags as the prior work [58, 85] did. However, we found two types of bugs, *i.e.*, `accept-invalid` bugs and `reject-valid` bugs, which were never reported by the prior approaches [58, 85].

4.5.1 Accept-invalid bugs. In an `accept-invalid` bug, a compiler wrongly accepts an invalid program.

Figure 5a shows a program. `clang` accepts it, but it is invalid, since the `noexcept` methods in Lines 3 and 7 do not match. The bug was already fixed after we reported it. Figure 5b shows a program. `clang` developers once marked the bug report as `invalid`, but later changed it to `reopen`. A `clang` developer left a message: "*After some discussion on the core reflector, ... a strict reading of [dcl.fct]p6 suggests that it is not in the set, so GCC is correct to reject.*" Figure 5c shows another `clang` bug. A developer confirmed that they did not check static data: "*We diagnose the mismatch for variable templates but strangely not for static data members of class templates*". Figure 5d shows a `gcc` bug. `gcc` accepts it, but a developer confirmed that the code is invalid: "*[expr.reinterpret.cast] p11 covers casting to a reference type, and is only allowed when the source is a glvalue, so this is invalid*". Figure 5e shows an ill-formed lambda expression. A developer confirmed that `icc` and `msvc` also wrongly accept the program. Figure 5f shows another `gcc` bug. `gcc` accepts the program, but a developer confirmed that `clang`, `icc`, and `msvc` all correctly reject the program.

4.5.2 Reject-valid bugs. In a `reject-valid` bug, a compiler wrongly rejects a valid program.

Figure 6a shows a `clang` bug. It is a valid code, since a fold-expression is a primary-expression. The bug is already fixed after we reported it. Figure 6b shows a valid program, but `clang` rejects it. A developer indicated that this may be a known bug, since a related source file has a `FIXME` in its code comments. Figure 6c shows a `gcc` bug. `gcc` rejects a valid program. Its developers already fixed the bug, after we reported it. Figure 6d shows another `clang` bug. `clang` developers confirmed that it is a bug, and `icc` has the same problem. However, they did not identify which libraries shall be changed to fix the bug. Figure 6e shows a `gcc` bug. `gcc` determines that `B` is not declared, but it is. Figure 6f shows another `gcc` bug. A developer confirmed that `icc` and `msvc` also wrongly reject the program.

4.5.3 Other bugs. Besides the `accept-invalid` bugs and `reject-valid` bugs, LeRe detected a few other bugs.

Figure 7a shows an `ice` bug in `gcc`. `gcc` crashes when it compiles the program. The buggy code does not fully check the conditions, before it handles anonymous aggregates. The bug was fixed, after we reported it. Figure 7b shows a `wrong-code` bug in `gcc`. In Line 12,

```
1 extern "C" void abort();
2 static int i;
3 int *p = &i;
4 int main(){
5  int i;
6  {
7    extern int i;
8    i = 1;
9    *p = 2;
10   if (i == 2) abort ();
11 }
12 return 0;}
```

Figure 8: The program is extracted from a bug report of 2007, and it still triggers a bug in the trunk version of gcc.

gcc considers 42 and -1 as const `A&` and `A&&`, instead of integer values. A developer confirmed that "*Maybe the object is being copied by an implicitly-defined copy constructor, but that's meant to be deleted and overload resolution should have used the move constructor*". Figure 7c shows a link failure. A flag is no longer set, so a check is disable, after `gcc3.2`. As a result, latter versions have the problem. After we reported the bug [14], gcc developers have fixed the bug four months later.

4.6 Threat to Validity

The threat to internal validity includes the manual process to determine real bugs. We once reported dozens of bugs in a day, and our reported bugs were not related. Compiler developers suspected that we were not real users. In that day and the follow-up days, they marked many of our bug reports as invalid or duplicated. To reduce the threat, we list all our bug reports on our website, so others can inspect our found bugs based on their own expertise. The threat to internal validity also includes random test programs, since researchers can submit such programs in their bug reports. Still, the impacts shall be minor, since most bugs are reported by real programmers, even if their programs can be reduced. The threat to external validity includes the generality of LeRe to other languages. Although this is a limitation to our tool, it reveals that our research direction has many research opportunities.

5 LESSON ON DIFFERENTIAL TESTING

We notice that some of our reported bugs were marked as `invalid`. Differential testing has long been introduced to detect compiler bugs [34, 85], but most researchers did not report their found invalid bug reports in their papers. Although Le *et al.* [58, 59] reported several invalid bug reports, they did not analyze why their bug reports were considered as invalid. We carefully analyzed our invalid bug reports. Although they are marked as invalid, we find that they detect useful difference between compilers. Figure 9 shows the cases where this oracle fails:

Figure 9a shows a bug report. Lines 5 and 6 of its program declare two items with the same name. A `clang` developer explains that the validity of this program is undefined: "*CWG does not have a consensus position on the desired validity of this example*". Figure 9b shows a `clang` report. In C++, `_Atomic` is undefined: "*_Atomic is a reserved identifier (per [lex.name]), with no defined meaning in C++, so we can define it to mean whatever we want*". Figure 9c shows a `clang` bug report. `clang` accepts it, but `gcc` produces an error message on Line 5. A developer explained that their flexible array member extensions are different: "*Clang's flexible array member extension isn't the same as GCC's, and allows this*". Figure 9d shows an invalid gcc bug report. `clang` complains that in Line 2 the requested alignment is dependent but the declaration is not dependent. A developer explained "*This is an explicit extension which GCC supports. Clang might not want to support this extension but GCC does*". Figure 9e shows a gcc bug report. `clang` accepts it, but gcc produces an error message on Line 1. A gcc developer explained "*We don't have attribute ext_vector_type (we have vector_size). Gcc warns about it*". Figure 9f shows an invalid gcc bug report. A gcc developer explained the difference: "*The standard says passing non-trivial types through varargs is conditionally-supported so an implementation can either support it, or reject it with a diagnostic*".

Although these differences are false alarms, it is useful to warn programmers the differences. If programmers blindly believe that their code will produce the same results across compilers, they can introduce many latent bugs. Programmers shall understand the differences, when they are switching from one compiler to the other. Most of our found false alarms are undefined in the C++ standard, which highlights the importance of drafting better standards.

Indeed, all test oracles have limitations. For example, a test oracle says that programs shall not crash [30], but even this widely believed test oracle can fail. For example, if a pirated key is sent to a game server, the game can crash as designed, but this crash is not considered as a bug.

Some programmers complain that bug detection tools have high false alarms [57]. Besides incomplete specifications [60] and the limitations of detection techniques [57], we identify that test oracles can also introduce false alarms. However, our findings shall not be interpreted as an excuse to only reject papers. Indeed, as a response of such reviewer comments, many researchers start to hide their false alarms with various tricks. In our humble view, this is not a constructive way to build the knowledge, and introduces long-term harms to the research community. Instead, we list the false alarms in our paper, so follow-up researchers can understand the inevitable false alarms and locate where to make improvements.

6 RELATED WORK

Empirical studies on bug reports and compiler testing. Researchers [23, 52, 61] conducted various empirical studies to understand the characteristics of bug reports. Researchers also conducted empirical studies to compare compiler testing approaches [39] or to analyze the characteristics of compiler bugs [71]. Our work is not an empirical study, but an experience paper. In our research context, the technology includes differential testing and extracting code snippets from SE documents, and our target problem explores a new source for extracting compiler test inputs. Our experience report shows that our problem is practical important for compiler developers, since they have confirmed more than one hundred bugs. Our lessons provide insights on the limitation of differential testing, and these findings can motivate exciting research on this issue.

Bug report analysis. Bettenburg *et al.* [31] analyze factors that contribute to a good bug report. A hot research line is to identify duplicate bug reports [67, 74, 76], and the other hot research line is to assign bug reports [23, 24, 82]. Tian *et al.* [78] build the priority list of bug reports. Bachmann *et al.* [29] and Wu *et al.* [81] build

```
namespace N{
 int i;
}
void f (){
 using N::i;
 using N::i;
}
```

(a) The C++ standard (http://wg21.link/cwg36) does not have a consensus postilion on this program.

```
struct AtomicInt
{
 _Atomic(int) Atomic;
};
AtomicInt GlobalAtomic0 = {0};
constexpr AtomicInt GlobalAtomic1 = {0};
```

(b) clang and gcc handle _Atomic differently, since it is a reserved identifier without defined meaning in C++.

```
struct x0{
 x0 () = default;
};
struct x1{
 x0 x2[];
 void x3 (){
   x1 ();
 }
};
```

(c) The two compilers have different flexible-array-member extension.

```
template <typename T> void Fun(T A) {
 typedef int __attribute__((__aligned__(A))
     ) T1;
 int k1[__alignof__(T1)];
}
```

(d) gcc supports the explicit extension, but clang does not.

```
typedef __attribute__((ext_vector_type(4)))
     int vi4;
const int &r = vi4(1).x;
```

(e) clang implements the ext_vector_type attribute, but gcc does not.

```
struct Foo {
 Foo() {}
 Foo(const Foo&) {}};
void f(...);
void g() {
 Foo foo;
 f(foo);
 void (*fp)(...) = f;
 fp(foo);}
```

(f) The C++ standard does not defines behaviors clearly.

(a) https://bugs.llvm.org/show_bug.cgi?id=38144; (b) https://bugs.llvm.org/show_bug.cgi?id=37922; (c) https://bugs.llvm.org/show_bug.cgi?id=38265; (d) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86578; (e) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86477; and (f) https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86562

Figure 9: False alarms

the links between bug reports and bug fixes. Zhou *et al.* [92] locate buggy files of a given bug report. LeRe extracts test inputs from bug reports, complementing these approaches.

7 CONCLUSION AND FUTURE WORK

To generate more test programs for compilers, researchers have proposed approaches that randomly generate programs or mutate existing programs. Compared with random code, it is more interesting to generate real test programs, since it is feasible to trace their authors and it is reasonable to use both valid and invalid real programs. However, an arbitrary source file is often ineffective to detect compiler bugs. Based on our observations on the bug fix process of compiler bugs, we identify bug reports as a new source for enriching compiler testing. To illustrate the benefits of the new source and to support our study, we implement LeRe that extracts test programs from bug reports and uses differential testing to detect compiler bugs. With LeRe, our experience report shows that more than one hundred new bugs were found, and two types of our found bugs are never reported in the prior approaches. Furthermore, while the prior approaches found bugs in C programs, LeRe is able to handle other programming languages.

Some next research opportunities are as follows.

Detecting more types of compiler bugs. With our extracted test programs, the prior compiler testing approaches [36, 37, 43] can have more test inputs, and more types of compiler bugs can be thus detected. First, a bug report often contains an original test program and its reduced version. Taking original programs as inputs, it is feasible to detect more `wrong-code` bugs as the prior work [58, 85] did, and comparing original programs with their reduced versions can motivate better techniques to reduce test programs [26, 55, 69]. Second, we can try more compiler flags to detect optimization bugs as other researchers [58, 59, 85] do. Finally, Le *et al.* [58] believe that it is feasible to extend their approach to support C++. After they extend their tool, it is feasible to feed our extracted programs to Le *et al.* [58, 59] as their seeds. The combination can detect more bugs in the C++ component, especially for the optimization phases.

Exploring other sources to extract effective real programs for compiler testing. Although bug reports of compilers provide a rich source of real test programs, the compilers for a new programming language may not have many bug reports. For such compilers, we envisage other sources to extract real test programs. For example, Yan *et al.* [83] show that programmers can workaround compiler bugs, and such changes can be identified from commits of code repositories. In particular, if bug reports mention compiler bugs, it is feasible identify the code fragments from corresponding commits [62]. After such programs are mined, many follow-up approaches shall be extended to handle real programs. For example, Le *et al.* [58] complain that it is difficult to reduce real programs. When reducing programs, most approaches [38, 40, 68, 79] assume that the file that triggers compiler bugs is a single file. To enable such approaches for real programs, researchers shall narrow compiler bugs to a single file.

Generating test cases from the bug reports of other applications. For example, some recent approaches (*e.g.*, [84]) can detect similar modules across different projects. Furthermore, a recent study [75] asks students to detect bugs of an Android application, based on the known bug reports of its related applications, and these students have detected some bugs with similar symptoms in this way. Bettenburg *et al.* [33] proposed an approach that extracts structural information from bug reports. Researchers have explored how to generate test inputs based on core dumps [80], runtime logs [86], and natural language descriptions [49]. These approaches can be useful to extend LeRe.

ACKNOWLEDGMENTS

We appreciate reviewers for their insightful comments. This work is sponsored by the CCF-Huawei Innovation Research Plan No. CCF2021-admin-270-202111.

REFERENCES

[1] 2020. The test suite guide of llvm. https://llvm.org/docs/TestSuiteGuide.html. (2020).
[2] 2021. Partial specialization halfway accepted after instantiation. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=32505. (2021).
[3] 2021. An test case that originates from a bug report. https://github.com/gcc-mirror/gcc/blob/master/gcc/testsuite/g%2B%2B.dg/template/partial8.C. (2021).
[4] 2022. The call for papers of ASE2022. https://conf.researchr.org/track/ase-2022/ase-2022-research-papers. (2022).
[5] 2022. Clang 10531. https://bugs.llvm.org/show_bug.cgi?id=10531. (2022).
[6] 2022. Clang 38235. https://bugs.llvm.org/show_bug.cgi?id=38235. (2022).
[7] 2022. Clang 38268. https://bugs.llvm.org/show_bug.cgi?id=38268. (2022).
[8] 2022. Clang 5134. https://bugs.llvm.org/show_bug.cgi?id=5134. (2022).
[9] 2022. Clang 5966. https://bugs.llvm.org/show_bug.cgi?id=5966. (2022).
[10] 2022. Clang 9989. https://bugs.llvm.org/show_bug.cgi?id=9989. (2022).
[11] 2022. GCC 31775. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=31775. (2022).
[12] 2022. GCC 59480. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=59480. (2022).
[13] 2022. GCC 86181. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86181. (2022).
[14] 2022. GCC 86208. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86208. (2022).
[15] 2022. GCC 86385. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86385. (2022).
[16] 2022. GCC 86502. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=86502. (2022).
[17] 2022. The ISO C++ standard. https://isocpp.org/std/the-standard. (2022).
[18] 2022. Languagetool. https://www.languagetool.org. (2022).
[19] 2022. OpenMandriva. https://www.openmandriva.org. (2022).
[20] 2022. PlumHall. http://www.plumhall.com/stec.html. (2022).
[21] 2022. SuperTest. http://www.ace.nl/compiler/supertest.html. (2022).
[22] 2022. The Microsoft C++ compiler. https://msdn.microsoft.com/en-us/library/ms235639.aspx. (2022).
[23] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proc. ICSE*. 361–370.
[24] John Anvik and Gail C Murphy. 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology* 20, 3 (2011), 10.
[25] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. 2011. Extracting structured data from natural language documents with island parsing. In *Proc. ASE*. 476–479.
[26] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. 2012. Content classification of development emails. In *Proc. 34th ICSE*. 375–385.
[27] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. 2010. Extracting source code from e-mails. In *Proc. 18th ICPC*. 24–33.
[28] Alberto Bacchelli, Michele Lanza, and Romain Robbes. 2010. Linking e-mails and source code artifacts. In *Proc. 32nd ICSE*. 375–384.
[29] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *Proc. ESEC/FSE*. 97–106.
[30] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.
[31] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proc. FSE*. 308–318.
[32] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate bug reports considered harmful ... really?. In *Proc. ICSM*. 337–345.
[33] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting structural information from bug reports. In *Proc. MSR*. 27–30.
[34] Abdulazeez S Boujarwah and Kassem Saleh. 1997. Compiler test case generation methods: a survey and assessment. *Information and software technology* 39, 9 (1997), 617–625.
[35] Preetha Chatterjee, Benjamin Gause, Hunter Hedinger, and Lori Pollock. 2017. Extracting code segments and their descriptions from research articles. In *Proc. MSR*. 91–101.
[36] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proc. ICSE*. 700–711.
[37] 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 *Proc. ICST*. 266–277.
[38] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2022. Compiler bug isolation via effective witness test program generation. In *Proc. ESEC/FSE*. 223–234.
[39] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proc. ICSE*. 180–190.
[40] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *Proc. ASE*. to appear.
[41] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *Comput. Surveys* 53, 1 (2020), 1–36.
[42] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2022. History-guided configuration diversification for compiler test-program generation. In *Proc. ASE*. 305–316.
[43] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proc. PLDI*. 197–208.
[44] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proc. ESEC/FSE*. 793–804.
[45] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proc. ICSM*. 1–10.
[46] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink. In *Proc. ICSE*. to appear.
[47] Barthélémy Dagenais and Martin P. Robillard. 2012. Recovering Traceability Links between an API and its Learning Resources. In *Proc. 34rd ICSE*. 47–57.
[48] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *Proc. ESEC/FSE*. 185–194.
[49] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proc. ISSTA*. 141–152.
[50] Eibe Frank. 2000. *Pruning Decision Trees and Lists*. Ph.D. Dissertation. Department of Computer Science, University of Waikato.
[51] Yoav Freund and Robert E. Schapire. 1996. Experiments with a new boosting algorithm. In *Proc. ICML*. San Francisco, 148–156.
[52] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proc. ICSE*. 495–504.
[53] Kenneth V. Hanford. 1970. Automatic generation of test cases. *IBM Systems Journal* 9, 4 (1970), 242–257.
[54] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proc. USENIX*. 445–458.
[55] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proc. ICSE*. 38–48.
[56] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. Crash consistency validation made easy. In *Proc. ESEC/FSE*. 133–143.
[57] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proc. ICSE*. 672–681.
[58] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proc. PLDI*. 216–226.
[59] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proc. OOPSLA*. 386–399.
[60] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *Proc. ASE*. 602–613.
[61] Zexuan Li and Hao Zhong. 2021. An empirical study on obsolete issue reports. In *Proc. ASE*. 1317–1321.
[62] Zexuan Li and Hao Zhong. 2021. Understanding code fragments with issue reports. In *Proc. ASE*. 1312–1316.
[63] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. In *Proc. PLDI*. 65–76.
[64] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing system virtual machines. In *Proc. ISSTA*. 171–182.
[65] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[66] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Proc. SASIMI 2012*. 48–53.
[67] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proc. ICSE*. 70–79.
[68] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proc. PLDI*. 335–346.
[69] Peter C. Rigby and Martin P. Robillard. 2013. Discovering Essential Code Elements in Informal Documentation. In *Proc. 35th ICSE*. 11.
[70] RP Seaman. 1974. Testing compilers of high level programming languages. *IEEE Compututer System and Technology* (1974), 366–375.
[71] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proc. ESEC/FSE*. 968–980.
[72] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proc. OOPSLA*. 849–863.
[73] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proc. ISSTA*. 294–305.
[74] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. ICSE*. 45–54.

[75] Shin Hwei Tan and Ziqiang Li. 2020. Collaborative Bug Finding for Android Apps. In *Proc. ICSE*. to appear.
[76] Ferdian Thung, Pavneet Singh Kochhar, and David Lo. 2014. DupFinder: integrated tool support for duplicate bug report detection. In *Proc. ASE*. 871–874.
[77] Yuan Tian, Nasir Ali, David Lo, and Ahmed E Hassan. 2016. On the unreliability of bug severity data. *Empirical Software Engineering* 21, 6 (2016), 2298–2323.
[78] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. 2015. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering* 20, 5 (2015), 1354–1383.
[79] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proc. ESEC/FSE*. 881–892.
[80] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proc. APLOS*. 155–166.
[81] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*. 15–25.
[82] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2015. Towards Effective Bug Triage with Software Data Reduction Techniques. *IEEE Transactions on Knowledge and Data Engineering* 27, 1 (2015), 264–280.
[83] Aoyang Yan, Hao Zhong, Daohan Song, and Li Jia. 2022. The Symptoms, Causes, and Repairs of Workarounds in Apache Issue Trackers. In *Proc. ICSE*. to appear.
[84] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in Web App model inference. In *Proc. ICSE*. 186–197.
[85] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proc. PLDI*. 283–294.
[86] Tingting Yu, Tarannum S Zaman, and Chao Wang. 2017. DESCRY: reproducing system-level concurrency failures. In *Proc. ESEC/FSE*. 694–704.
[87] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Automatically learning patterns for self-admitted technical debt removal. In *Proc. SANER*. 355–366.
[88] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proc. PLDI*. 347–361.
[89] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proc. ICSE*. 1133–1144.
[90] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proc. OOPSLA*. 803–816.
[91] Hao Zhong, Suresh Thummalapenta, and Tao Xie. 2013. Exposing behavioral differences in cross-language API mapping relations. In *Proc. ETAPS/FASE*. 130–145.
[92] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*. 14–24.