An Empirical Comparison of Compiler Testing Techniques*

Junjie Chen[1,2], Wenxiang Hu[1,2], Dan Hao[1,2†‡], Yingfei Xiong[1,2‡],
Hongyu Zhang[3‡], Lu Zhang[1,2], Bing Xie[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, EECS, Peking University, Beijing 100871, China
{chenjunjie,huwx,haodan,xiongyf,zhanglucs,xiebing}@pku.edu.cn
[3]Microsoft Research, Beijing 100080, China, honzhang@microsoft.com

ABSTRACT

Compilers, as one of the most important infrastructure of today's digital world, are expected to be trustworthy. Different testing techniques are developed for testing compilers automatically. However, it is unknown so far how these testing techniques compared to each other in terms of testing e ectiveness: how many bugs a testing technique can nd within a time limit.

In this paper, we conduct a systematic and comprehensive empirical comparison of three compiler testing techniques, namely, Randomized Di erential Testing (RDT), a variant of RDT | Di erent Optimization Levels (DOL), and Equivalence Modulo Inputs (EMI). Our results show that DOL is more e ective at detecting bugs related to optimization, whereas RDT is more e ective at detecting other types of bugs, and the three techniques can complement each other to a certain degree.

Furthermore, in order to understand why their e ectiveness di ers, we investigate three factors that in uence the e ectiveness of compiler testing, namely, e ciency, strength of test oracles, and e ectiveness of generated test programs. The results indicate that all the three factors are statistically signi cant, and e ciency has the most signi cant impact.

1. INTRODUCTION

Compilers are important because they are widely used in software development. Buggy compilers may lead to the unintended behaviors of developed programs, which may cause software failures or even disasters in safety-critical domains. Furthermore, compiler bugs make debugging more di cult because developers can hardly determine whether software failure is caused by the software they are developing or the compilers they are using. Therefore, guaranteeing the quality of compilers is critical.

However, it is very challenging to guarantee the quality of compilers. Although compilation theory and compiler design [1, 24, 8] have been thoroughly studied [10, 29, 32], in reality, compilers still contain bugs [12]. Like many other software systems, compiler testing su ers from the test-oracle problem [2]. That is, it is hard to determine the expected outputs of a compiler under test given some programs as the test inputs.

To automate compiler testing without oracles, several techniques have been proposed in the literature. Most of the current work on compiler testing is based on Randomized Di erential Testing (RDT) [18, 29, 30, 31, 26], which assumes that several comparable compilers are implemented based on the same speci cation and detects bugs by comparing the outputs of these compilers for the same test program. When these compilers produce di erent results, some of the implementations must contain bugs. Furthermore, as many compiler bugs reported in previous work are triggered by compiler optimizations [29, 13], a simple testing technique is to compare the outputs of one compiler at di erent optimization levels for the same test program. We call such a technique Di erent Optimization Levels (DOL), which is technically a variant of RDT. Recently, Le et al. [13] proposed a compiler testing technique called Equivalence Modulo Inputs (EMI), which generates a series of variants from an existing test program by guaranteeing the variants to be equivalent to the test program under a set of test inputs. It then detects bugs by comparing the outputs of the given test program and those of the variants. The above-mentioned work on compiler testing is widely recognized. For example, the work on EMI [13] received the Distinguished Paper award from PLDI 2014 and the work on RDT is widely used in actual compiler testing practice [29].

Although existing studies have investigated the e ectiveness of the compiler testing techniques, they either use only one compiler testing technique [29, 13] or focus on a small domain of dedicated compilers (i.e., OpenCL compilers) [16]. To our best knowledge, it is yet unknown which of the three techniques is more e ective in detecting general compiler bugs. Furthermore, because of the particularity of compilers, in the remainder of this paper, \test programs" refers to the inputs of compilers and \test inputs" refers to the inputs of test programs.

*We would like to acknowledge Zhendong Su and Chengnian Sun at UC Davis for their valuable and insightful comments on an early version of this paper. This work is supported by the National Basic Research Program of China (973) under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61421091, 91318301, 61332010, 61522201, 61225007, 61272089.

†Corresponding author.
‡Sorted in the alphabet order of the last names.

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884878

In this paper, we present a systematic and comprehensive empirical comparison of the three automated compiler testing techniques (RDT, DOL, and EMI). We implement the three techniques and apply them to two mainstream open-source C compilers (GCC and LLVM), which cover all C compilers used in existing studies of compiler testing [13, 29]. To investigate the effectiveness of the three compiler testing techniques, we qualitatively and quantitatively analyze the bugs detected by these techniques and derive our findings.

A key challenge in this empirical comparison is to measure the effectiveness of the compiler testing techniques. Ideally, compiler testing techniques should be measured in terms of the number of compiler bugs detected, but there is no means for us to measure the number of bugs directly. Previous work has used two methods to approximate the ideal measurement. Lidbury et al. [16] and Yang et al. [29] used the number of test programs that detect faulty behaviors in compilers. However, as will be shown in Section 5.1, this measurement is highly inaccurate because different test programs can trigger the same bug. Le et al. [13] asked compiler developers to identify the test programs that trigger the same bug, but this method does not scale well.

In this paper, we propose a new measurement: Correcting Commits. When a test program fails on an early version of the compiler, we check subsequent compiler commits, and determine which commits correct the bugs. By using the number of correcting commits to approximate the number of bugs, our measurement avoids many of the inaccuracies in previous studies.

Our empirical study also investigates three factors that influence the effectiveness of compiler testing. We study how the three techniques perform with respect to the three factors and the statistical impact of the three factors on the effectiveness of compiler testing. The three factors are:

- Efficiency: how many test programs can be tested given a fixed period of time?
- Strength of test oracles: given the same test programs, which technique detects more bugs?
- Effectiveness of generated test programs: EMI generates new variants from the original program. Do the variants help discover more bugs than randomly generated programs?

The major findings that we obtain from the empirical comparison are as follows.

- DOL is more effective at detecting optimization-related bugs and RDT is more effective at detecting optimization-irrelevant bugs.
- RDT can substitute EMI and DOL in detecting optimization-irrelevant bugs, but it can be substituted by DOL in detecting optimization-related bugs. Furthermore, both EMI and DOL cannot be substituted in detecting optimization-related bugs.
- With respect to the three factors, we find the following.
  - DOL is the most efficient technique, whereas EMI is the least efficient one.
  - RDT oracle is the strongest, whereas EMI oracle is the weakest.
  - The randomly generated programs are more effective than the variants generated by EMI.
  - The impact of all the three factors is statistically significant.
  - Efficiency has the most significant impact on the effectiveness of compiler testing, and the effectiveness of generated test programs has the least significant impact.

The main contributions of this paper can be summarized as follows:

- A systematic and comprehensive empirical comparison of automated compiler testing techniques and obtain many valuable findings.
- A new measurement, Correcting Commits, which better measures the effectiveness of compiler testing than previous measurements.

2. COMPILER TESTING TECHNIQUES

In this section, we introduce the three automated compiler testing techniques: RDT, DOL, and EMI.

2.1 Randomized Differential Testing

RDT [18, 29, 30, 31, 26] is a widely-used compiler testing technique, which addresses the test oracle problem in compiler testing using two or more comparable compilers that implement the same specification. Because these comparable compilers should produce the same results under the same set of test inputs, it is easy to determine which compilers contain bugs through voting. That is, if more than half of the compilers generate the same results, the results are regarded as the correct results.

Given a set of compilers under test, denoted as $\{C_1, C_2, \ldots, C_n\}$ where $n \geq 3$, the process for applying RDT to test these compilers is as follows. Each compiler C_i compiles a test program P and generates an executable E_i, where $1 \leq i \leq n$. For any given set of test inputs for P, denoted as I, these executables produce different results, denoted as $O_1, O_2, \ldots, O_n$. Because the compilers under test are designed to follow the same specification, their behaviors are expected to be the same. Therefore, RDT detects compiler bugs through the voting among O_1, O_2, ..., and O_n.

RDT is a well-known testing technique, which is applicable to both compilers and other complex software because it is effective and easy to implement. However, there is an important limitation of RDT. Because these compilers could be implemented in a similar manner or with some common source code, it is possible for most or all comparable compilers to produce the same wrong results under the same set of test inputs. In this scenario, RDT can hardly detect compiler bugs. Furthermore, if there is a new programming language and only one compiler is available for this programming language, RDT cannot be applied to test this compiler at all.

2.2 Different Optimization Levels

As mentioned before, we consider a variant of RDT: comparing compilation results under different optimization levels. That is, when a test program is respectively compiled under different optimization levels (e.g., -O0, -O1, -Os, -O2 and -O3 in GCC) and executed under the same set of test

inputs, it may produce di erent results, thus we know that the compiler under test contains bugs. For simplicity, in this paper we informally call those bugs detected through di erent optimization levels as \optimization-related bugs", and other bugs as \optimization-irrelevant bugs".

Similar to RDT, DOL uses one test program each time, thus it is easy to be implemented. Furthermore, DOL targets at only one compiler with di erent optimization levels, thus it can directly test a speci ed optimizing compiler.

2.3 Equivalence Modulo Inputs

EMI [13] is a newly proposed compiler testing technique, which addresses the test oracle problem through comparison between a test program and its variants whose behaviors are regarded as equivalent under a set of test inputs for this test program. In particular, for a test program, EMI identi es a set of statements that a ect its behavior given some test inputs, and constructs variants whose behaviors are equivalent to the behavior of the test program. EMI introduces a \Pro le and Mutate" strategy on the statements of the given test program and uses this strategy to generate variants.

For a compiler under test (denoted as C), the process for applying EMI to test this compiler is as follows. For any given test program P and its test inputs I, EMI rst generates some variants of P (denoted as Q_1, Q_2, ..., Q_m) through the following process. First, EMI identi es a set of statements in P unexecuted by I through dynamic analysis, and generates variants by randomly deleting statements within this set. Then the test program P and each variant Q_i ($1 \leq i \leq m$) are compiled by the compiler C, and thus the corresponding executables (denoted as E_p and E_i) are generated. Taking I as test inputs, these executables produce results, denoted as O_p and O_i ($1 \leq i \leq m$). Because the variants should behave equivalently to given program P under test inputs I, EMI detects the bugs in C by comparing each pair O_p and O_i, where $1 \leq i \leq m$.

Although EMI is more complex than RDT and DOL, EMI targets at only one compiler with/without di erent optimization levels, thus it can directly test any speci ed compiler and overcome the limitation of RDT and DOL.

3. MEASUREMENT: CORRECTING COMMITS

In this paper, we propose a new measurement: Correcting Commits. The new measurement is based on the analysis of commit history of the compiler under test. For any test program that triggers a bug of a compiler C whose commit version is x, we check subsequent commits of the compiler, and determine which commits correct the bug. To locate the commit that corrects the corresponding bug, our approach rst nds a commit of the compiler that has already corrected the bug, and then performs binary search between the commit with the bug and the commit without the bug.

In the repository, branches can be created and later merged back. However, in the commit history of the compilers used in this study, we observe only the creation of branches but not the merging of two branches. Therefore, we ignore merge and represent the commit history by a tree, which is illustrated by Figure 1 where nodes represent commits and edges represent the inheritance relation between commits.

To nd the correcting commit quickly, we identify the

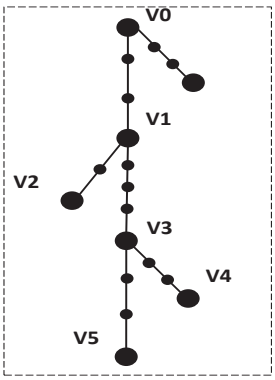


Figure 1: Tree Model for the Commit History

nodes that incur branches (e.g., $V0$ and $V1$) and terminate branches (e.g., $V2$ and $V4$), which are called key nodes in this paper, and use a tree structure with root node denoted as $Tree$, to represent a commit history consisting of only key nodes and their inheritance relation. In particular, the root node is also regarded as a key node and represents the version under test. Based on these key nodes, the commit history is divided into several segments (e.g., $V0 - V1$ and $V1 - V2$), each of which is denoted as linked list $LS(Vs, Ve)$, representing that there is an inheritance path from the key node Vs to another key node Ve. Given the commit history composing of commits $\{V0, V1, \ldots, Vm\}$, where $V0$ represents the compiler version under test and Vm represents its latest version or commit, and $PS=\{P_1, P_2, \ldots, P_n\}$ represents the set of test programs that trigger bugs, we use Algorithm 1 to calculate the number of commits correcting the bugs triggered by PS. We use a set Set to record the correcting commits. This algorithm repeats n times to nd the correcting commit for each test program P_i where $1 \leq i \leq n$. In particular, Lines 2-3 deal with the event that the bugs triggered by P_i are not corrected at all. Line 5 is to nd the correcting commit for P_i through the method $Find$ and add it to Set if this correcting commit has never been added to Set. Line 8 is to produce the number of correcting commits for PS.

Algorithm 2 presents the method $Find$, whose parameter $Tree$ refers to the commit history and P refers to the test program that triggers a bug. The variable $flag$ represents the correcting commit if there exists for P. Algorithm 2 is a depth- rst algorithm, which searches the tree from the root node so as to nd a key node (denoted as V') that corrects the bugs detected by P. If no commit in the history corrects the corresponding bugs, the return value is set to zero according to Lines 3 to 5; otherwise, the correcting commit is in the linked list $LS(V, V')$. In this case, this algorithm nds the speci c correcting commit in this linked list through the widely used binary search algorithm (implemented as $BinarySearch(LS(V, V'))$ in this paper) [23].

Based on this algorithm, for any set of test programs that trigger failing compiler behaviors, we can learn the number of correcting commits automatically. Being a measurement for compiler testing, the number of correcting commits tends to be more accurate than the number of test programs in approximating the number of bugs, since one correcting commit is often for xing one bug. That is, this newly proposed measurement may avoid some inaccuracies in previous studies, which will be further studied in Section 5.1.

Algorithm 1 Correcting Commits Measurement

```
1: for each i (1 ≤ i ≤ n) do
2:    if !Correct(P_i, V_m) then
3:       Set.add(0)
4:    else
5:       Set.add(Find(Tree, P_i))
6:    end if
7: end for
8: output(Set.size)
```

Algorithm 2 int Find(Tree,P)

```
1: V ← Tree
2: flag ← 0
3: if V.children() == NULL then
4:    return 0
5: end if
6: for each i (0 ≤ i ≤ V.children.length) do
7:    V' ← V.children[i]
8:    if Correct(P, V') then
9:       return BinarySearch(LS(V, V'))
10:   else
11:      flag = Find(V', P)
12:      if flag! = 0 then
13:         return flag
14:      end if
15:   end if
16: end for
17: return 0
```

4. STUDY DESIGN

4.1 Subjects

We use two mainstream open-source C compilers GCC[1] and LLVM[2] for the x86_64-Linux platform as the subjects of the empirical study, which are C compilers used in existing studies of compiler testing [13, 29]. Because it is impossible to determine which compiler is wrong when applying RDT to only two compilers, we add another C compiler ICC[3] to our empirical study; this is a commercial optimizing compiler from Intel with high reliability. The ICC compiler serves as the golden compiler[4] in the RDT experiment.

In particular, in the empirical study, we use GCC 4.4.3 and LLVM 2.6. We choose these versions because they are based on the same C99 speci cation. Therefore, they can be viewed as comparable compilers, thus satisfying the requirements of RDT.

4.2 Measurement of Testing Effectiveness

In the empirical study, we use the number of test programs that reveal compiler bugs and the number of correcting commits obtained by the approach described in Section 3, to measure the e ectiveness of a compiler testing technique. In particular, in our study, all the bugs detected by the three techniques have been xed by correcting commits. If no confusion is caused, we use the number of bugs to directly refer to the number of correcting commits in the remainder of this paper.

[1]http://gcc.gnu.org/.

[2]The LLVM project is a collection of compiler and toolchain techniques, which is accessible at http://llvm.org/. To be consistent with previous work on compiler testing, we also use LLVM to represent the compiler used in LLVM, which is mainly Clang.

[3]https://software.intel.com/en-us/c-compilers.

[4]The golden compiler is used to determine which compiler is wrong if the two compared compilers produce di erent results.

4.3 Test Programs

To compare the e ectiveness of the three compiler testing techniques, we use CSmith [29] to generate a set of test programs and use this set of test programs as the inputs to the compilers. CSmith is a widely used test generation tool for C compilers, which randomly generates C programs without unde ned behavior. In this study, we use CSmith 2.2.0 with its default con guration. The test programs generated by CSmith do not require external inputs, and its output is a checksum of their non-pointer global variables at the end of the test-program execution. That is, we do not need to generate test inputs for these test programs.

In particular, for EMI, besides these test programs, variants are generated when applying the standard \Pro le and Mutate" strategy (described in Section 2.3) to the test programs. In particular, for each test program generated by CSmith, we generate eight variants[5] by deleting the unexecuted statements randomly following Le et al. [13].

4.4 Process

To reduce the in uence of test programs in the comparison of compiler testing techniques, we prepare a xed sequence of randomly generated test programs so that the compared compiler testing techniques use the same test programs in this empirical study.

Given a subject compiler, we run each of the three compiler testing techniques for 90 consecutive hours by taking these test programs as inputs, and record the test programs with which the corresponding compiler testing technique detects bugs. To compare the three compiler testing techniques, we calculate the number of test programs that detect compiler bugs and the number of compiler bugs detected by each compiler testing technique. The process of evaluating these compiler testing techniques in this study is as follows.

First, we apply RDT to test the two compilers GCC and LLVM. Following the prior work [29], we compile each test program with GCC and LLVM under the same optimization level, which contain -O0, -O1, -Os, -O2 and -O3. We compiler each test program with ICC without any optimizations. We then execute the corresponding executables. If the test results produced by GCC and LLVM are di erent, at least one compiler is wrong. At this time, by comparing the results produced by ICC, GCC, and LLVM, we determine in which compiler the corresponding test program detects a bug. That is, we regard ICC as the golden compiler when GCC and LLVM produce di erent results.

Second, we apply EMI to test the two compilers. We execute the original test program and the eight variants, and compare the result of each variant with the result of the original program. Following the prior work [13], we use ve optimization levels (i.e., -O0, -O1, -Os, -O2 and -O3) to compile the original test program and its variants, respectively. We determine whether the compiler under test contains bugs by comparing the output of the original test program and each of its variants. If the output of the original test program is di erent from that of the variant, this pair of programs detects a bug in the compiler.

Third, we apply DOL to test the two compilers. We compile each test program using the same compiler under dif-

[5]The number of variants is recommended by their paper [13].

ferent optimization levels (i.e., -O0, -O1, -Os, -O2 and -O3), and compare the execution results produced by each optimization level (i.e., -O1, -Os, -O2 and -O3) with that produced by -O0 (i.e., no optimization at all).

Our empirical study was conducted on a workstation with eight-core Intel Xeon E5620 CPU (2.4GHz) with 24G memory, and Ubuntu 12.04.5 operating system.

5. RESULTS AND ANALYSIS

As measurement is the base of an empirical study, in this section, we rst quantitatively compare the newly proposed measurement (i.e., Correcting Commits) with the mostly used measurement (i.e., the number of test programs) on their accuracies (in Section 5.1). That is, this study aims to tell which measurement is more accurate in evaluating compiler testing.

Based on the new measurement, we compare the three compiler testing techniques in terms of their e ectiveness, substitutability, and the characteristics of detected bugs (in Section 5.2). That is, this study aims to tell which compiler testing technique should be applied in practice.

The inherent di erence of the three compiler testing techniques lies in the test programs and oracles. For test programs, RDT and DOL do not have any novelty, but EMI de nes a new type of test programs, i.e., variants generated based on existing test programs. For test oracles, RDT, DOL, and EMI use various mechanisms, i.e., relation between compilers/optimizations/programs. Furthermore, as compiler testing usually takes a long time, the e ciency issue, i.e., how many test programs are used in compiler testing in a given period of time, is also an important internal factor that in uences how a compiler testing technique performs. Therefore, we further study the impact of these inherent factors (in Section 5.3) so as to learn the secret why a compiler testing technique performs well.

Besides, many C compiler bugs reported in the literature are optimization-related bugs. Therefore, we further investigate how optimization levels in uence compiler testing (in Section 5.4).

The experimental data is available at the project webpage[6].

5.1 Measurement Comparison

As mentioned in Section 1, previous work measured the e ectiveness of compiler testing techniques by the number of test programs that detect faulty behavior in compilers, but it may be inaccurate because di erent test programs may trigger the same bug. In this section, we evaluate the accuracy of their measurement.

We manually check ve commits of GCC, each of which xes only one GCC bug. Table 1 presents the number of test programs that trigger the ve bugs, respectively. From this table, the distribution of the number of test programs that trigger each bug is extremely uneven, because 877 test programs trigger the 5th bug and only one test program triggers the 2nd bug. That is, the measurement that uses the number of test programs can obtain very inaccurate results. However, during the development of compilers, one commit cannot correct hundreds of bugs. In other words, the number of correcting commits is much closer to the ideal measurement (i.e., the number of detected bugs) than the number of test programs that trigger bugs. Therefore, this is evidence for the necessity of using the number of correcting commits rather than the number of test programs as a measurement in compiler testing.

[6]http://emponcc.github.io/

Table 1: Number of Test Programs Triggering Bugs

Bug	1st	2nd	3rd	4th	5th
Test Programs Triggering Bugs	17	1	1	1	877

Furthermore, Table 2 presents the number of bugs detected by the three compared techniques and the number of test programs that trigger bugs during the 90 hours of experiments. From this table, we nd that the number of test programs can lead to the wrong conclusion in measuring the e ectiveness of compiler testing techniques, because DOL detects the largest total number of bugs under our measurement but its number of the test programs that trigger these bugs is smaller than that of RDT. This is another evidence for the necessity of using the number of correcting commits rather than the number of test programs as a measurement in compiler testing.

Finding 1:
The number of test programs that trigger bugs is an inaccurate measurement in compiler testing, and thus the number of correcting commits is necessary as a measurement.

5.2 Compiler Testing Technique Comparison

5.2.1 Effectiveness

• Number of Detected Bugs

From Table 2, the total number of bugs detected by DOL (31) is larger than that by EMI (16) and RDT (26). Among the three compiler testing techniques, EMI detects the smallest number of bugs. Furthermore, DOL detects the largest number of GCC bugs, whereas RDT detects the largest number of LLVM bugs. That is, RDT and DOL seem to be more e ective than EMI.

Table 2: Effectiveness Comparison

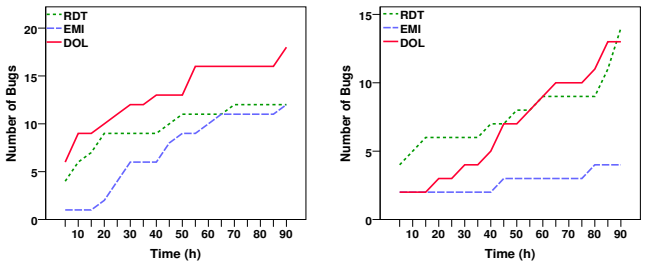
Compilers	Detected Bugs			Test Programs that Trigger Bugs		
	RDT	EMI	DOL	RDT	EMI	DOL
GCC	12	12	18	422	492	954
LLVM	14	4	13	801	6	54
TOTAL	26	16	31	1,223	498	1,008

• Number of Bugs Detected per Ten Hours

Because compiler testing is costly, compiler testing techniques are expected to detect bugs as early as possible. Therefore, besides the total number of detected bugs, we calculate the number of bugs detected per ten hours to measure the cost-e ectiveness of the three compiler testing techniques. Figure 2 presents the increased number of GCC and LLVM bugs detected by the compared techniques with time increasing. From Figure 2(a), the number of GCC bugs detected by DOL is always larger than that by RDT and EMI, and the increasing rate of DOL and EMI are obvious larger than that of RDT. From Figure 2(b), the number of LLVM bugs detected by either RDT or DOL is much larger than that by EMI and their increasing rates are also larger than EMI. In summary, DOL is superior to RDT and EMI in GCC, and RDT and DOL are superior to EMI in LLVM.

• Time Spent on Detecting the First Bug

It is desirable for a testing technique to reveal the rst bug early so that developers can start debugging early. Therefore, we further compare how much time each compiler testing technique spends on detecting the rst bug, and the re-



(a) GCC (b) LLVM

Figure 2: Increased Number of Detected Bugs per Ten Hours

sults are listed in Table 3. From this table, DOL requires the least time to nd the rst GCC bug, whereas RDT requires the least time to nd the rst LLVM bug.

Table 3: Time to Detect the First Bug (seconds)

Compilers	RDT	EMI	DOL
GCC	116	978	84
LLVM	34	17,571	11,363

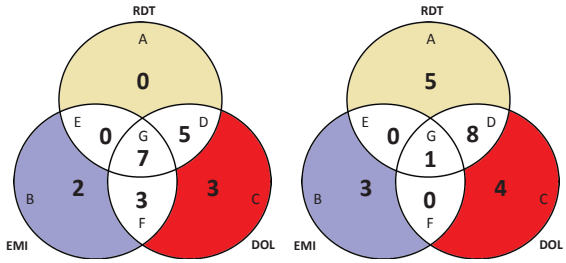
Finding 2:
DOL seems to be the most e ective one at detecting GCC bugs and RDT seems to be the most e ective one at detecting LLVM bugs in our experiments.

5.2.2 *Substitutability*

In the preceding analysis, we investigate the three techniques based on one criterion, the total number of detected bugs. However, there is another important criterion to compare the three techniques: whether the bugs detected by one compiler testing technique can be detected by the others. In other words, we need to study the number of bugs that can be detected by only one compiler testing technique, which are called **unique bugs** for the corresponding technique in this paper. The more unique bugs a compiler testing technique detects, the less substitutable the technique tends to be. On the contrary, if a compiler testing technique cannot detect any unique bug, it can be substituted by other compiler testing techniques because all its detected bugs can also be detected by the latter techniques.

To visualize the number of unique bugs detected by each technique, we use Venn Diagrams [28] to represent the sets of bugs detected by these techniques, as shown in Figure 3. The three circles represent the sets of bugs detected by each compiler testing technique. Each circle is divided into several areas, each of which represents the set of bugs detected by the corresponding technique, and it is marked by a letter from A to G for reference. For example, the area marked by \A" represents the set of unique bugs only detected by RDT. The number in each area represents the number of elements in the corresponding set. For example, in Figure 3(a), 0 in Area A denotes that zero unique bug is detected by RDT alone, whereas 5 in Area D denotes that ve bugs are detected by both RDT and DOL, but not by EMI.

From this gure, the number of GCC unique bugs detected by RDT is zero, thus the combination of EMI and DOL can completely substitute RDT in detecting GCC bugs. In fact, Figure 3(a) shows that all GCC bugs detected by RDT are also detected by DOL, but ve GCC bugs detected by RDT are not yet detected by EMI. Therefore, only DOL can completely substitute RDT in detecting GCC bugs. However, from Figure 3(b), RDT has the largest number of LLVM



(a) GCC (b) LLVM

Figure 3: Number of Unique Bugs

unique bugs (i.e., ve). Moreover, the number of unique GCC bugs detected by DOL is the largest of all (i.e., three), and the number of unique LLVM bugs detected by DOL is four. Therefore, DOL cannot be substituted by RDT and EMI. Similarly, EMI cannot be also substituted by RDT and DOL although it has the least e ectiveness and cost-e ectiveness (described in Section 5.2.1). In summary, DOL detects the most GCC unique bugs and RDT detects the most LLVM unique bugs, and EMI can complement the other two techniques because it can detect some unique bugs.

Finding 3:
DOL seems to be more e ective at detecting GCC unique bugs and RDT seems to be more e ective at detecting LLVM unique bugs. Furthermore, when detecting GCC bugs, RDT can be substituted by DOL completely.

5.2.3 *Optimization-Related/Irrelevant Bugs*

In Section 2.2, we indicate that DOL is designed to detect optimization-related bugs. That is, if a bug can be detected by DOL, the bug is an optimization-related bug. Therefore, we compile and execute the test programs that trigger bugs by RDT and EMI, and determine whether the test programs also trigger the bugs by DOL. If a bug detected by RDT or EMI is also detected by DOL, the bug is an optimization-related bug; otherwise, it is viewed as an optimization-irrelevant bug.

Table 4 presents the number of optimization-related bugs and optimization-irrelevant bugs detected by each compiler testing technique. From this table, all the bugs detected by EMI are optimization-related bugs, and the bugs detected by RDT contain both types of bugs. In particular, DOL detects the most optimization-related bugs, whereas RDT detects the most optimization-irrelevant bugs. Therefore, DOL seems to be more e ective than RDT and EMI at detecting optimization-related bugs, whereas RDT seems to be more e ective than EMI and DOL at detecting optimization-irrelevant bugs. Furthermore, because all the GCC bugs detected by the three techniques are optimization-related bugs, GCC may have more optimization-related bugs than optimization-irrelevant bugs, and this observation is consistent with the conclusion from Le et al. [13].

In particular, we can complement the conclusion from Figure 3 based on the results in this section. In Figure 3, ve bugs in Area A of Figure 3(b) are optimization-irrelevant bugs, and all the remaining bugs are optimization-related bugs. Therefore, RDT can completely substitute EMI and DOL at detecting optimization-irrelevant bugs and DOL can completely substitute RDT at detecting optimization-related bugs. EMI cannot be substituted by other techniques in detecting optimization-related bugs because it detects ve

unique optimization-related bugs. Similarly, DOL cannot be substituted by other techniques in detecting optimization-related bugs because it detects seven unique optimization-related bugs.

Table 4: Comparison on Optimization-Related Bugs and Optimization-Irrelevant Bugs

Bugs	Optimization-Related			Optimization-Irrelevant		
	RDT	EMI	DOL	RDT	EMI	DOL
GCC	12	12	18	0	0	0
LLVM	9	4	13	5	0	0
TOTAL	21	16	31	5	0	0

Finding 4:
DOL seems to be more e ective at detecting optimization-related bugs, whereas RDT seems to be more e ective at detecting optimization-irrelevant bugs. Furthermore, GCC may have more optimization-related bugs than optimization-irrelevant bugs.

5.3 Impact of Inherent Factors

5.3.1 Efficiency

In this subsection, we rst investigate the impact of e - ciency, which is the number of test programs actually used in compiler testing in a given period of time. Table 5 presents the number of test programs compiled and executed during the experimental period (90 hours) for testing GCC and LLVM by RDT, EMI and DOL, respectively. In particular, the number of test programs for EMI in this table refers to the number of original test programs.

This table indicates that, during the same period of time, DOL uses the largest number of test programs, whereas EMI uses the smallest. Therefore, DOL and EMI are the most and least e cient, respectively. The observation is as expected. For each test program, RDT needs 11 compilations because it needs three compilers, and it uses ve optimization levels for GCC and LLVM and one optimization level for ICC. EMI needs 45 compilations because it generates eight variants for each original test program and also uses ve optimization levels for the compiler under test, and DOL needs ve compilations because it uses ve optimization levels for the compiler under test.

Table 5: Number of Test Programs Used During the Experimental Period

Compilers	RDT	EMI	DOL
GCC	27,990	4,794	62,552
LLVM	27,990	5,040	64,385

Furthermore, based on the preceding analysis, DOL detects the largest number of bugs, which contains many unique bugs, and EMI detects the smallest number of bugs. That is, the more e cient the technique is, the larger the number of detected bugs is. Therefore, e ciency is an important factor that in uences the e ectiveness of compiler testing.

Finding 5:
DOL is the most e cient technique, whereas EMI is the least e cient one.

5.3.2 Strength of Test Oracles

Similarly, in this subsection we investigate the strength of test oracles: given the same test programs, which technique detects more bugs. This is the second internal factor that in uences the e ectiveness of compiler testing.

In fact, all three techniques are proposed to address the test oracle problem. RDT determines whether a bug is detected by comparing the results produced by di erent compilers, and DOL determines whether a bug is detected by comparing the results produced by di erent optimization levels of the same compiler. In particular, EMI consists of two novel components, including a variant generation component and test oracle, which determines whether a bug is detected by comparing the results of the test program and its variants produced by the same compiler. In order to investigate the strength of test oracles, we use the same programs including original test programs and their corresponding variants generated by EMI to test GCC and LLVM by the three test oracles.

Table 6 presents the results of investigating the strength of test oracles. This table shows that, when using the same programs including original test programs and their variants, RDT detects the largest number of bugs and unique bugs, whereas EMI detects the smallest number of bugs and unique bugs. In particular, when testing GCC, RDT detects the same number of bugs and unique bugs with DOL. Therefore, RDT oracle is the strongest whereas EMI oracle is the weakest. When testing GCC, RDT oracle is as strong as DOL oracle.

In practice, if combining the variant generation component of EMI and RDT/DOL oracle, the e ectiveness of the combinations may be much better than that of EMI. Therefore, the strength of test oracles is another factor that in uences the e ectiveness of compiler testing techniques.

Table 6: Strength of Test Oracles

Compilers	Detected Bugs			Unique Bugs		
	RDT	EMI	DOL	RDT	EMI	DOL
GCC	18	12	18	0	0	0
LLVM	16	4	10	6	0	0
Total	34	16	28	6	0	0

Finding 6:
RDT and DOL oracles are stronger than EMI oracle, and RDT oracle is not weaker than DOL oracle.

5.3.3 Effectiveness of Generated Test Programs

In this subsection, we investigate the impact of the third factor: e ectiveness of the generated test programs.

In this empirical comparison, there are two types of generated test programs, namely randomly generated programs and variants generated by EMI. In order to investigate the e ectiveness of generated test programs, we use the same number (i.e., 18,900) of randomly generated programs and variants generated by EMI to test compilers using RDT and DOL, and record the number of bugs detected by them, respectively. We do not include EMI in this experiment, because EMI cannot be applied when there is only the randomly generated programs.

Table 7 presents the results of comparing the quality of randomly generated programs and variants, where \Random" represents the randomly generated programs. In particular, the number of unique bugs means the number of bugs detected by the combination of a technique and a type of programs but not detected by the combination of this technique and the other type of programs.

Table 7 shows that, when RDT uses randomly generated programs, it detects more bugs and unique bugs than when it uses variants; when DOL uses randomly generated programs, it detects more bugs and unique bugs than when it uses variants. Therefore, randomly generated programs detect bugs more easily than variants. That is, RDT/DOL

that uses randomly generated programs may be a better choice than RDT/DOL that uses variants in practice. However, the role of variants cannot be replaced because when RDT/DOL uses variants, they also detect some unique bugs. Therefore, the e ectiveness of generated test programs is also an important factor that in uences the e ectiveness of compiler testing techniques.

Table 7: Effectiveness of Generated Test Programs

Compilers	RDT		DOL	
	Random	Variant	Random	Variant
GCC Bugs	11	8	11	8
LLVM Bugs	9	6	4	2
GCC Unique Bugs	6	3	6	3
LLVM Unique Bugs	5	2	3	1

Finding 7:
The variants generated by EMI are less e ective than randomly generated programs by CSmith, but variants can serve as a complement to randomly generated programs.

5.3.4 Statistical Significance

In this subsection, we analyze the impact of the three factors on compiler testing, and analyze which factor has more signi cant impact. In statistics, this question can be regarded as a multivariable linear regression problem, namely, analyzing the impact of three independent variables on a dependent variable (i.e., the number of detected bugs). In particular, e ciency is a quantitative variable, but the test oracle and generated test program are qualitative variables, thus we use dummy variables to represent them. There are three test oracles, namely RDT oracle, EMI oracle and DOL oracle. There are two types of generated test programs, namely randomly generated programs and variants generated by EMI. We test GCC and LLVM under di erent combinations of test oracles and test programs. In particular, we use the same randomly generated programs or variants.

Table 8 presents the results of statistics analysis. Due to the lack of the combination of EMI oracle and randomly generated programs, we perform the statistical analysis in two steps. First, we analyze the impact of the three factors, but the factor test oracles have only two values (i.e., RDT oracle and DOL oracle). Second, we analyze the impact of e ciency and the strength of test oracles, but test oracles have three values this time, RDT oracle, EMI oracle and DOL oracle. In this table, Rows 2 to 4 present the results of the rst statistical analysis, and Rows 5 to 7 present the results of the second statistical analysis. The second column presents the coe cient of regression equation, and the last column presents the signi cance value whose signi cant level is set to 0.05. In particular, the absolute value of the coe cient represents the degree of correlation, and its sign represents positive correlation or negative correlation, and the signi cance value re ect whether the factor has signi cant impact on the e ectiveness of compiler testing.

First, from this table, all the signi cance values are less than 0.05, which means that the impact of the three factors on the e ectiveness of compiler testing is statistically significant. Then, based on the results of the rst statistical analysis, the absolute value of the coe cient of e ciency is the largest one, and that of \Program (Random v.s. Variant)" is the smallest one, thus e ciency has the most signi cant impact on the e ectiveness of compiler testing, whereas the e ectiveness of the generated test program has the least signi cant impact on it. Furthermore, the coe cient of \Test Oracle (RDT v.s. DOL)" is -0.386, which is smaller than 0, thus RDT oracle is stronger than DOL oracle. Similarly, the coe cient of \Program (Random v.s. Variant)" is -0.138, which means that randomly generated programs are more effective than variants generated by EMI, con rming the conclusion of Section 5.3.3. Besides, based on the results of the second statistical analysis, e ciency also has the most signi cant impact on the e ectiveness of compiler testing. Furthermore, the coe cient of \Test Oracle (EMI v.s. RDT)" is 0.589, which means that RDT oracle is stronger than EMI oracle, and the coe cient of \Test Oracle (EMI v.s. DOL)" is 0.284, which means that DOL oracle is also stronger than EMI oracle, con rming the conclusion of Section 5.3.2.

Table 8: Statistics

Factors	Coefficient	Sig
Test Oracle (RDT v.s. DOL)	-0.386	0.041
Program (Random v.s. Variant)	-0.138	0
Efficiency	0.824	0
Test Oracle (EMI v.s. RDT)	0.589	0
Test Oracle (EMI v.s. DOL)	0.284	0.002
Efficiency	0.777	0

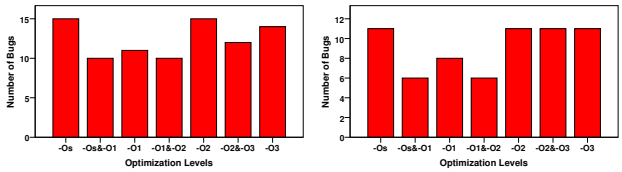
Finding 8:
All the three factors have statistically signi cant impact on the e ectiveness of compiler testing, where e ciency has the most signi cant impact and the e ectiveness of generated test programs has the least signi cant impact.

5.4 Influence of Optimization Levels

As many of the detected bugs are related to optimization according to the preceding experiment results, we further analyze the in uence of optimization levels on triggering compiler bugs. Here we present only the analysis results of DOL, which is more e ective at detecting optimization-related bugs, due to space limitation.

Figure 4 presents the number of bugs detected with di erent optimization levels (i.e., -O1, -Os, -O2 and -O3), respectively, where the horizontal axis lists these optimization levels. -O1 is the lowest optimization level, which implements the basic optimization operations when compiling. From -O1 to its right, -O3 is the highest optimization level, and from -O1 to its left, -Os is higher than -O1 because -Os is a special optimization level and adds some space optimization operations based on -O1. In particular, -Oi&-Oj represents the number of intersection of bugs detected at -Oi and bugs detected at -Oj, where -Oi and -Oj are two adjacent optimization levels. Furthermore, the vertical axis represents the number of detected bugs.

In Figure 4(a), the numbers of bugs detected at -Os and -O2 are the largest of all optimization levels, and in Figure 4(b), the numbers of bugs detected at -Os, -O2 and -O3 are the largest of all optimization levels. It is easy to take for granted that the higher the optimization level is, the larger the number of detected bugs is, because a higher optimization level contains all the optimization operations from the lower optimization levels. However, the assumption is invalid from Figure 4(a) because -O3 does not detect the largest number of bugs, although it is the highest optimization level among -O1, -O2 and -O3. On the other hand, the number of bugs detected with two adjacent optimization levels is usually smaller than the lower one. That is, when using a higher optimization level, the bugs detected at a lower optimization level may disappear. We suspect the reason for this observation to be the combination of opti-



(a) GCC (b) LLVM

Figure 4: Influence of Optimization Levels

mization options at certain optimization level. This conclusion indicates that it is necessary to test a compiler under di erent optimization levels.

Finding 9:
As some compiler bugs are only triggered by lower optimization levels alone, it is necessary to test a compiler with various optimization levels.

6. THREATS TO VALIDITY

6.1 Threats to Internal Validity

The threats to internal validity mainly lie in the implementation of the compiler testing techniques. First, because the implementation of EMI, especially its \Pro le and Mutate" strategy, is not available, we implemented this technique using the same tool (LibTooling Library of Clang[7]) as Le et.al [13] used. Furthermore, the number of variants and the deletion probability may also a ect the e ectiveness of EMI, we will explore their e ect in future. Second, when implementing RDT, we simply regard ICC as a golden compiler, although it may happen to contain the same bug as GCC or LLVM. To reduce this threat, in our study, we turned o the optimization features of ICC when compiling test programs using ICC, because it is safer when optimizations are disabled [29].

6.2 Threats to External Validity

The threats to external validity mainly lie in subjects and test programs.

Subjects. In this study, we use two C compilers (i.e., GCC and LLVM) as subjects. Nevertheless, the two compilers already cover all the C compilers used in existing studies of compiler testing [29, 13]. In addition, since CSmith was used to nd bugs in GCC and LLVM before 2009 [29], the compilers used in our paper may have already acquired a certain amount of immunity to test programs generated by CSmith, which may have an impact on the e ectiveness of these techniques. However, all the three techniques are fed with the same test programs generated by CSmith, thus the threat may not be serious.

Test Programs. In this empirical study, we use only the test programs generated by CSmith as the inputs to compilers. In the previous work on compiler testing [13, 16], in addition to these generated programs, compiler test suites and existing open-source projects are used in compiler testing. We do not use them in our study because (1) compiler test suites are ine ective in detecting bugs in released compilers [13] (2) open-source projects usually contain code with unde ned behavior [16], and it is di cult to determine whether the detected abnormalities are bugs.

6.3 Threats to Construct Validity

[7]http://clang.llvm.org/docs/LibTooling.html

The threats to construct validity mainly lie in the con gurations of RDT and EMI, the measurement of experimental results, the randomness in our experiments and the testing period of our experiments.

Configurations of RDT and EMI. In this study, when testing compilers using RDT and EMI, we use ve optimization levels of the compilers under test, and EMI generated eight variants for each test program. These con gurations may in uence the e ectiveness of these techniques. In order to reduce this threat, we set these con gurations of RDT and EMI according to prior work [29, 13].

Measurement. In our study, we measure the e ectiveness of compiler testing techniques based on the number of correcting commits. Although this automatic approach facilitates a systematic empirical study on compiler testing, the identi cation results may not be very precise. That is, the test programs that detect di erent bugs may be regarded as detecting the same bug in our study, if these bugs are corrected in the same commit. To verify the severity of this threat, we check all the commits that we used in our study and nd that 11 commits state explicitly which bugs are xed by them, and each commit of them xes only one bug. That is, the threat may not be a serious threat in our study. In the future, we will use those known bugs to provide complementary evaluation in order to reduce the threat of our new measurement.

Randomness. The randomness (i.e., randomly generated test programs) in our experiment may have an impact on our conclusions. To reduce this threat, we adopt an extended period of testing time and repeat the experiments. For all the experiments except Table 3, we adopt an extended period of testing time (i.e., 90 hours on each testing technique) to o set the randomness threat because in our experimental settings, adopting an extended period of testing time is equivalent to repeating the experiments. For the results in Table 3, we repeat this experiment to reduce the threat of randomness. Based on ve runs, the conclusion from Table 3 still holds in all runs, and we select one of these runs as the representative.

Testing Period. Some testing techniques have been applied to extremely long periods of testing. For example, EMI has been used in testing the trunk version of C compilers for 11 months [13]. Here is a question whether our results obtained from 90 hours of testing can be generalized to extremely long periods or not. We do not have a de nitive answer to the question yet. However, some trends can be observed from Figure 2. On the increasing rate of detected bugs from GCC, RDT is obviously slower than EMI and DOL, while no obvious di erence can be told between EMI and DOL. In the case of LLVM, EMI is obviously slower than DOL and RDT, while no obviously di erence can be told from DOL and RDT.

7. DISCUSSION

7.1 About Optimization

Based on Section 5.3, e ciency has the most signi cant impact on the e ectiveness of compiler testing. The higher the e ciency is, the larger the number of detected bugs is. That is, to improve the e ectiveness, compiler testing techniques should be designed to deal with as many test programs as possible in any given period of time. We further analyzed the three compiler testing techniques and found

that optimization levels influence their efficiency a lot. The more optimization levels are used, the fewer test programs can be used in any given period of time. That is, using many optimization levels seems to harm the effectiveness of compiler testing. However, due to the conclusion in Section 5.4, using all the optimization levels is preferable considering bug detection. That is, there is a dilemma in the choice of optimization levels. Therefore, in the future, we need to find some combinations of optimization levels to maximize the effectiveness of compiler testing.

7.2 About Inherent Factors of Compiler Testing Techniques

Besides the widely concerned test oracles, EMI presents a new type of test programs, which are variants generated based on existing test programs. Based on the preceding analysis, these variants are less effective than randomly generated programs, but the former can still detect some unique bugs. That is, being a new type of test programs, variants can serve as a complement to randomly generated test programs. Furthermore, we find both RDT and DOL oracles are stronger than EMI, using RDT oracle or DOL oracle as guide, generating test programs aiming at specific test oracle may be an effective way of improving the effectiveness of compiler testing. Besides, as efficiency is more important for compiler testing than generated test programs and oracles, compiler testing techniques should be designed to support the execution of more test programs in any given period of time. In the future, we need to explore how to balance these factors to improve the effectiveness of compiler testing through the generation or optimization of test programs.

7.3 About Practical Usage

Our findings suggest that, in practice we could use all the three techniques to test compilers in the order of DOL, RDT, and EMI, because DOL is the most efficient technique and RDT is the second efficient technique, and both RDT and EMI can detect unique bugs.

8. RELATED WORK

The most relevant work are the studies about compiler testing. Yang et al. [29] and Le et al. [13] investigated the effectiveness of RDT and EMI, respectively, and both of them analyzed compiler bugs in detail, but they focused on only one compiler testing technique. Lidbury et al. [16] evaluated the effectiveness of RDT and EMI, but they focused on a small domain of dedicated compilers | OpenCL compilers. In our work, we present a systematic and comprehensive empirical comparison of all the three compiler testing techniques (i.e., RDT, DOL and EMI) on two mainstream open-source C compilers (i.e., GCC and LLVM) and propose a new measurement: Correcting Commits.

Besides, test program generation is also an important aspect in compiler testing. When applying differential testing to compiler testing, one of the main challenge lies in the generation of test programs. According to the survey conducted by Boujarwah and Saleh [4], a large amount of work focuses on generating test programs to facilitate compiler testing. McKeeman [18] proposed to construct new C programs by adding or deleting some elements (in different levels) to any given C program. To our knowledge, his work is the first to emphasize the importance of avoiding undefined behavior in generated C test programs for testing C compilers. However, his work cannot generate C programs without undefined behaviors. Bazzichi and Spadafora [3] proposed to generate programs through a tabular description of the source language. Hanford [9] proposed to use a PL/1 grammar to generate programs randomly. Kalinov et al. [11] proposed an approach to generating compiler test suite automatically based on several coverage criteria. Zhao et al. [31] developed an integrated tool (JTT) driven by test specification to automatically generate programs to test UniPhier, an embedded C++ compiler. Nagai et al. [19, 20] proposed to generate C programs, which contain randomly generated arithmetic expressions and successfully avoid undefined behaviors, to test C compilers' arithmetic optimization. Lindig [17] proposed to use randomly generated C programs to test the consistency of C compilers. His tool (Quest) is type-directed and can be controlled by the user, rather than depends on control flow and arithmetic. Sheridan [26] proposed to test C99 compilers by comparing the behaviors of randomly generated programs using C99 compilers and using pre-existing tools. In addition, Sauder [25] proposed to test the logic of the COBOL compiler by placing random variables in the data sections of the programs. Palka et al. [21] proposed to randomly generate lambda terms to test an optimizing compiler, which mainly addresses the type-correct issue in generation. Callahan et al. [5] proposed to test the effectiveness of vectorizing compilers by a collection of 100 Fortran loops. Recently, CSmith [29, 7, 22] is proposed and implemented to randomly generate C programs without undefined behavior for testing C compilers.

Unlike random differential testing, Le et al. [13] proposed to generate some equivalent variants for each test program and detect bugs by comparing their behaviors. Similarly, Tao et al. [27] proposed to detect bugs by comparing the behaviors of several programs whose metamorphic relation is equivalence relation. Recently, Le et al. [14] proposed an advanced EMI to detect compiler bugs by using more mutation operations for test programs and by introducing Markov Chain Monte Carlo (MCMC) optimization to explore the search space. Furthermore, Le et al. [15] proposed to detect bugs of link-time optimizers by randomized stress-testing, and reported 37 bugs. Besides, in order to detect compiler bugs earlier, Chen et al. [6] proposed a test-vector based approach to prioritize test programs for compilers.

9. CONCLUSION

This work is a systematic and comprehensive empirical study that compares different compiler testing techniques: RDT, DOL, and EMI. From our study, we obtain some interesting findings. For testing compilers containing optimization-related bugs, DOL is more effective, and for testing compilers containing other types of bugs, RDT is more effective. In addition, there are three factors that influence the effectiveness of compiler testing, namely, efficiency, strength of test oracles and effectiveness of generated test programs, and their impacts are statistical significant. In particular, DOL is the most efficient technique, whereas EMI is the least efficient one; RDT oracle is the strongest, whereas EMI oracle is the weakest; the randomly generated programs are more effective than variants generated by EMI.

10. REFERENCES

[1] A. V. Aho. *Compilers: Principles, Techniques and*

Tools (for Anna University), 2/e. Pearson Education India, 2003.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507{525, 2015.

[3] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, (4):343{353, 1982.

[4] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: A survey and assessment. *Information and software technology*, 39(9):617{625, 1997.

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

[6] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. Test case prioritization for compilers: A text-vector based approach. In *Proceedings of the 9th International Conference on Software Testing, Verification and Validation*, 2016.

[7] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197{208. ACM, 2013.

[8] Y. Ding, J. Mei, and H. Cheng. Design and implementation of java just-in-time compiler. *Journal of Computer Science and Technology*, 15(6):584{590, 2000.

[9] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242{257, 1970.

[10] W. Harrison. Introduction to compiler construction. 2013.

[11] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Coverage-driven automated compiler test suite generation. *Electronic Notes in Theoretical Computer Science*, 82(3):500{514, 2003.

[12] A. Kossatchev and M. Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10{19, 2005.

[13] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, pages 216{226. ACM, 2014.

[14] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 386{399. ACM, 2015.

[15] V. Le, C. Sun, and Z. Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 327{337. ACM, 2015.

[16] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core complier fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 65{76, 2015.

[17] C. Lindig. Random testing of C calling conventions. In *Proceedings of the 6th international symposium on Automated analysis-driven debugging*, pages 3{12. ACM, 2005.

[18] W. M. McKeeman. Di erential testing for software. *Digital Technical Journal*, 10(1):100{107, 1998.

[19] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies*, pages 48{53, 2012.

[20] E. Nagai, A. Hashimoto, and N. Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies*, pages 88{93, 2013.

[21] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 91{97. ACM, 2011.

[22] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 335{346. ACM, 2012.

[23] R. L. Rivest and C. E. Leiserson. *Introduction to algorithms*. McGraw-Hill, Inc., 1990.

[24] H. Samet. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 annual conference*, pages 492{497. ACM, 1976.

[25] R. L. Sauder. A general test data generator for COBOL. In *Proceedings of Spring Joint Computer Conference*, pages 317{323. ACM, 1962.

[26] F. Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475{1488, 2007.

[27] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 17th Asia Pacific Software Engineering Conference*, pages 270{279. IEEE, 2010.

[28] J. Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 10(59):1{18, 1880.

[29] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, pages 283{294, 2011.

[30] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proceedings of the 3rd International Conference on Quality Software*, pages 20{23. IEEE, 2003.

[31] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test*, pages 36{43. IEEE, 2009.

[32] G. Zhu, L. Xie, and Z. Sun. Nuapc: A parallelizing compiler for c++. *Journal of Computer Science and Technology*, 12(5):458{469, 1997.