Front. Comput. Sci., 2020, 14(1): 1–20
https://doi.org/10.1007/s11704-019-8231-0

RESEARCH ARTICLE

Compiler testing: a systematic literature analysis

Yixuan TANG[1], Zhilei REN[1], Weiqiang KONG[1], He JIANG (✉)[1,2,3]

1 School of Software, Dalian University of Technology, Dalian 116024, China
2 Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116000, China
3 School of Computer Science & Technology, Beijing Institute of Technology, Beijing 100081, China

© Higher Education Press and Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract Compilers are widely-used infrastructures in accelerating the software development, and expected to be trustworthy. In the literature, various testing technologies have been proposed to guarantee the quality of compilers. However, there remains an obstacle to comprehensively characterize and understand compiler testing. To overcome this obstacle, we propose a literature analysis framework to gain insights into the compiler testing area. First, we perform an extensive search to construct a dataset related to compiler testing papers. Then, we conduct a bibliometric analysis to analyze the productive authors, the influential papers, and the frequently tested compilers based on our dataset. Finally, we utilize association rules and collaboration networks to mine the authorships and the communities of interests among researchers and keywords. Some valuable results are reported. We find that the USA is the leading country that contains the most influential researchers and institutions. The most active keyword is "random testing". We also find that most researchers have broad interests within small-scale collaborators in the compiler testing area.

Keywords software engineering, compiler-theory and techniques, literature analysis, collaboration network, bibliometric analysis

1 Introduction

Compilers are important infrastructure tools in software development, which provide syntax and semantics analysis for programs, as well as code optimization to accelerate software upgrades. For example, the security engineering group at Microsoft utilizes compilers to prioritize code review [1]; the maintenance engineers at Hewlett-Packard improve the quality of code by removing compiler diagnostics in software systems [2].

However, compilers may also contain bugs, and in fact quite many bugs are reported for widely-used compilers such as GCC and LLVM [3]. Buggy compilers make a source program optimized or translated into a wrong executable module, which may behave differently from the expected behavior determined by the semantics of the source program. Once this happens, it can result in disastrous software failures especially in safety-critical domains. For instance, a bug in the compiler of HAL/S had even caused the failure of the NASA Shuttle software. Even worse, developers with little knowledge about compiler bugs customarily debug the software they are developing rather than the compilers they are using, which makes compiler bugs more difficult to be found [4, 5]. Therefore, guaranteeing the quality of compilers is a critical issue.

Compiler testing is one of the most important ways to guarantee the quality of compilers. According to the previous studies, there are three issues to be addressed: how to generate adequate test cases to test compilers, how to find the test oracles to determine whether a test case triggers bugs, and how to reduce these test cases. Furthermore, two challenges are to be addressed. First, since the inputs of compilers are complex programs with furcated syntax structures and rigorous content constraints, undefined behaviors of lan-

Received July 1, 2018; accepted November 9, 2018

E-mail: hejiang@ieee.org

guage specification make the first issue and the third issue be a challenge [6]. Second, since compiler testing lacks test oracles to determine whether the outputs of compilers are semantic equivalent with the programs before they are compiled [7], the test oracle problem makes the second issue be a challenge.

During the past decades, a great number of researchers have proposed different approaches for addressing the above issues. Some successful random test case generators have been implemented to facilitate compiler testing [8–10], such as Orion [5], Csmith [6, 11, 12], Quest [13, 14], randprog [15], and JTT [16]. All of them can automatically generate abundant test programs for compilers without undefined behaviors. Simultaneously, various compiler testing techniques have been proposed to mitigate the test oracle problem, such as differential testing [4, 17], random testing [10, 18], Equivalence Modulo Inputs (EMI) [5], mutation testing [19], and metamorphic testing [20,21]. By employing the above testing techniques, a large number of compiler bugs can be detected. In addition, several reducers have been developed to minimize the test cases, such as Berkeley Delta, C-Reduce [12], and CL-Reduce [22]. Thus, a set of small and valid test cases that trigger the same bugs as original ones can be reported to developers.

However, as the number of related papers increases, there are few efforts to systematically identify, analyze, and classify the influential researchers, the state-of-the-art testing technologies, the collaborations among authors, and the co-occurrence of keywords, which results in an obstacle to characterize and understand compiler testing. In this study, we employ a systematic and comprehensive literature analysis framework to overcome the obstacle. First, we perform an extensive search to identify papers related to compiler testing, and extract the most important information from papers for the consequent analysis, such as the title, the keywords, and the author(s). Then, we conduct a bibliometric analysis to identify the most influential authors and papers, as well as the widely-used compiler testing technologies, so as to present an external overview of the compiler testing area. Last, we construct three collaboration networks to analyze the communities of authors and keywords, which can present internal evidence on the influential authors and hot topics in this area.

The major contributions of this paper are summarized as follows:

- We conduct a bibliometric analysis for compiler testing literature. The results show that the USA is the most influential country with a large number of excellent researchers and institutions in the compiler testing area. In addition, various types of compilers are tested, ranging from C++, Java to Pascal, whereas C compilers draw much attention from academia.
- We combine association rule mining and collaboration analysis to construct three networks, including the co-authorship network, the author co-keyword network, and the keyword co-occurrence network. The results show that most researchers have broad interests in the compiler testing area. These researchers distribute in several scattered communities. The keywords “test case generation”, “automated testing”, and “random testing” frequently co-occur in compiler testing.

The paper is structured as follows. Section 2 illustrates the challenges and the corresponding solutions in compiler testing. We demonstrate the components of literature analysis framework in Section 3. Then, Section 4 shows the findings from bibliographic and collaboration analyses. Section 5 provides an overview of related work. Section 6 concludes our paper and discusses the future direction.

2 Background of compiler testing

In this section, we briefly introduce the challenges and solutions to the three issues in compiler testing.

2.1 General compiler testing process

Compilers can transform the source program written in high-level language into language-independent machine code, and different compilers can transform the source program into distinct binaries under various build environments [23]. The process of transformation is called compilation which can be divided into three parts, i.e., frond end, middle end, and back end. In the frond end, the program can be transformed into intermediate code after the lexical analysis, syntactic analysis, and semantic analysis, in which the structure and the static semantic correctness of the program are verified. Then, in the middle end, the quality of intermediate code can be improved by machine-independent optimizers. Last, the code generator creates an executable file for the target machine according to the optimized intermediate code in the back end.

In most cases, each part of transformation may contain bugs, thus comprehensive tests should be conducted to guarantee the quality of compilers [6]. The general process of compiler testing is illustrated in Fig. 1, including three main issues. The first issue is the test case generation. The grammar

of language is guided to generate test cases and the expected outputs. Several useful tools such as Quest and Csmith can randomly generate abundant test cases for testing compilers. In the second issue, test cases as inputs of the compiler under test are executed, and the actual outputs are obtained. By employing different testing methods, such as differential testing, random testing, and metamorphic testing, the actual outputs are compared against the expected outputs. For example, in differential testing, a test case can be compiled under a golden reference compiler and a test compiler. The expected output is the behavior of the golden compiler, and the actual output is generated by the test compiler with the same test case input. If there is any difference, a bug manifests in the compiler under test. The last issue is to reduce test cases which can trigger compiler bugs. Several reducers can be applied to minimize test cases, such as Berkeley Delta and C-Reduce. Once the size of a test case is small enough, the bug can be reported to developers for analyzing the test alarms and further fixing [24]. However, each issue remains challenges that should be addressed. We present the challenges and some solutions to these challenges in the following subsections.

2.2 Test case generation issue

We illustrate the challenge in the test case generation issue and the solutions in this subsection.

There are several commercial test suites to test the quality of compilers, such as PlumHall, SuperTest, GNU Compiler Collection, and AC-TEST. Other test suites such as ACVC test suite, CppTestCase, Pascal Validation Suite, and COBOL validation tests are also employed by researchers for testing compilers. However, it is theoretically impossible to guarantee the correctness of compilers within a finite test suite. Actually, there are still many bugs in widely-used compilers, such as GCC and LLVM.

Random test case generation is an effective way to generate abundant test cases. Due to the reason that different languages are based on distinct language specifications, generating valid test cases that satisfy the corresponding language grammars is a much more difficult issue. In the case of C language, undefined behaviors make this issue a challenge. Undefined behaviors, such as zero division, signed overflow, and invalid pointer, may result in false positives. In other words, bugs are triggered by erroneous test case structures or erroneous data, rather than the compiler under test. Since possible undefined behaviors of C language may cause unexpected results and terminating execution, test cases must are free from these undefined behaviors.

The Purdom' algorithm [25] is an early prominent algorithm to generate test cases based on grammar rules, and has been extended to other test case generation approaches [26, 27]. Then, gaussian elimination [28] is applied to an industry example to test Fortran90D compiler. In addition, an ASM-based montages framework is proposed to generate test cases for mpC parallel programming language compiler [29], and find a lot of inconsistent places in the Montages specifications, as well as bugs in the compiler. After that, a tool named Quest can randomly generate test cases without undefined behaviors focusing on testing the consistency of C compilers. Randprog, another random C program generator, aims at detecting bugs in compiling accesses to volatile objects. JTT, an integrated tool, is driven by test specification

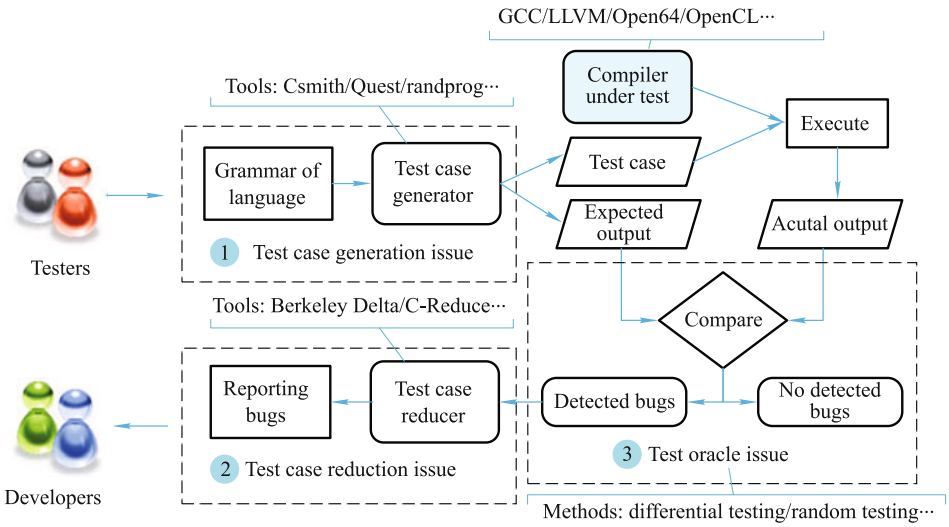


Fig. 1 General process of compiler testing

to automatically generate test cases for UniPhier compiler. Subsequently, Csmith extends and adapts Randprog to find bugs in C compilers, utilizing random C programs with complex control flow and data structures, such as pointers, arrays, and structs. Furthermore, CLsmith [8] has been proposed for many core compiler testing based on Csmith. However, neither Csmith nor CLsmith generates test programs for floating point test, which remains a challenge in the further test case generation.

More recently, Epiphron tools [4] targeted compiler warning bugs support nearly all the language structures of the C language. Other semantics and skeleton equivalent test cases are generated based on metamorphic testing and skeletal program enumeration (SPE) [30] respectively, to accelerate compiler testing. As so far, abundant test cases have been prepared to feed into compilers. Simultaneously, the expected outputs of these test cases should be collected.

2.3 Test oracle issue

In this subsection, we illustrate the challenge in the test oracle issue and the solutions to the challenge.

Given a test case to a compiler under test and a test input to the test case, the task to distinguish the expected and correct behavior of the test case from the potential incorrect behavior is called the "test oracle problem" [31]. However, the challenge is that it is difficult to determine whether the observed behavior is correct, because the expected behavior is difficult to be accurately described. In the literature, several approaches have been proposed to mitigate this issue. We categorize these approaches into two groups, namely the differential testing and the metamorphic testing.

Differential testing needs two or more compilers under the same specification to determine whether there is a bug by comparing the behaviors of these compilers given the same test cases as inputs. There are three strategies to implement differential testing, i.e., cross-compiler strategy [9], cross-optimization strategy [18], and cross-version strategy [4]. Cross-compiler strategy detects bugs by comparing the behaviors produced by different compilers; cross-optimization strategy compares the behaviors of different optimizations implemented in a single compiler, whereas cross-version strategy uses different versions of a single compiler to determine whether there is a bug. However, to the best of our knowledge, there are only a few formal verification compilers that can be used as a golden reference compiler for testing compilers, because of the difficulty of formal verification problem [32,33]. As a result, differential testing has its weakness when new programming languages are involved.

Notably, metamorphic testing introduces an alternative view on differential testing. If the behaviors of a set of semantically equivalent test cases dissatisfy the metamorphic relations, there is a bug manifests in the compiler under test. The advantages of metamorphic testing are that the approach can not only mitigate the test oracle problem, but also can be regarded as an effective complement to differential testing, especially when there are no available reference compilers. Furthermore, equivalence modulo input (EMI) which is derived from metamorphic testing adopts the equivalence relation under a set of oracle data as the metamorphic relation. The key insight behind EMI is to compare the results of source test case and its equivalent variants under the same oracle data to determine whether there is a bug in a compiler. Any detected deviant behavior on the same oracle data indicates a bug in the compiler. In fact, EMI has three instantiations, i.e., Orion, Athena [34], and Hermes [35]. Orion stochastically prunes program statements in dead regions. Athena utilizes Markov Chain Monte Carlo optimization to guide both code deletions and insertions in dead regions, and Hermes allows mutations in both live and dead regions to help more thoroughly stress test compilers. An empirical study [7] shows that different testing approaches are effective at detecting distinct types of compiler bugs. Cross-optimization strategy is more effective at detecting optimization-related bugs, and cross-compiler strategy can substitute EMI and Cross-optimization strategy in detecting optimization-irrelevant bugs. It is time consuming to test software, test case prioritization is a challenging task to accelerate software testing [36], especially in compiler testing [37].

2.4 Test case reduction issue

We present the challenge in the test case reduction issue and the corresponding solutions in this subsection.

To report a compiler bug, a test case that triggers the bug must be as small as possible because it is more difficult to reproduce due to the lengthy bug reports with diverse sentences and large size of test case [38]. In most cases, test cases are manually reduced which is laborious and time-consuming. Automatic test case reduction is required to help minimize test cases before reporting them to compiler developers. However, in the case of C language, undefined behaviors make this issue a challenge, because the test case should be free from undefined behaviors during the reduction process, and the reduced test case must trigger the same bug as the original one.

There are several reducers to automatically reduce test cases, including Berkeley Delta, C-Reduce, and CL-reduce. Berkeley Delta is based on delta debugging algorithm which reduces test cases at line granularity. C-Reduce is a state-of-the-art tool for reducing C programs which refers to abundant static and dynamic analyses to avoid undefined behaviors. Subsequently, C-Reduce is extended to CL-reduce which provides test case reduction for OpenCL kernels. Another approach adopts top-down minimization and bottom-up minimization algorithms alternately to reduce a tree structure constructed by arithmetic expressions until there is no space to minimize any more [39]. As a result, a test case with thousands of lines of code can be reduced to a few lines. However, all these reduction approaches only support single-file program reduction, whereas multiple-file programs reduction and real-world projects reduction still require further efforts.

Conclusion The compiler testing area includes three crucial issues, i.e., the test case generation issue, the test oracle issue, and the test case reduction issue. In order to address these three issues, two challenges need to be avoided, i.e., the undefined behaviors in test cases and the test oracle problem. In the literature, several approaches and tools are proposed to address these challenges. In order to investigate which approaches and tools are frequently employed when testing compilers, we conduct a bibliometric analysis, and present the results in Section 4.

3 Framework

The whole framework consists of three components, i.e., the dataset, the bibliometric analysis, and the collaboration analysis, as shown in Fig. 2. First, we construct a dataset containing the most important information of papers related to compiler testing in the dataset component. Then, the bibliometric analysis component provides three modules to present an overview of compiler testing. Last, we constructs three networks in the collaboration analysis component to present the internal evidence on collaborations between researchers and their interests. We detail each component of the framework in the following subsections.

3.1 Dataset

To construct the dataset, we refer to the processes of review study to find relevant published papers in journals and conference proceedings. We search three major online academic search engines, i.e., IEEE Xplore, ISI Web of Science (WoS), and Scopus. These search engines are widely accepted in review studies [40,41], and support advanced search. Then, we define a search string "compiler **AND** (test **OR** bug)", and limit the search within titles, abstracts, and keywords for paper selection. We do not limit a specific published time or journal/conference when conducting the searching. Therefore, the papers in our initial dataset are published before February 2018.

Since the focus of this paper is on compiler testing, many papers that target compiler verification and other software testing are included in our searching results. Thus, it is necessary to define comprehensive inclusion/exclusion criteria to select only the papers that provide evidence supporting for compiler testing.

For the inclusion criteria, we include the:

- Research papers that describe at least one compiler testing technology.
- Cases studies and surveys of compiler testing experiences.
- Papers of reference lists that are relevant to compiler testing.

For the exclusion criteria, we exclude the:

- Papers that are not published in English.

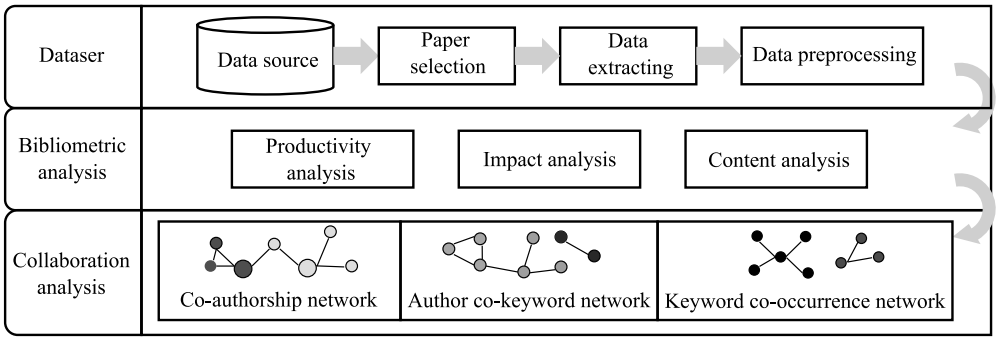


Fig. 2 The components of framework

- Resources of papers that are not available online.
- Short papers that are less than four pages.
- Papers that are duplications.
- Papers that are not related to compiler testing.

With the above search string, we find 6,731 papers in our initial dataset. We conduct the paper selection process, and present the collection of the number of papers after performing each criterion in parentheses as shown in Fig. 3. First, we check their titles to remove duplicates, and obtain 4,776 papers. Second, we excluded those papers that are less than four pages, and are not written in English. After applying this step, 711 papers are filtered. Then, we check the titles, keywords, and abstracts to eliminate irrelevant papers. In other words, only a paper describing the solutions to at least one issue in the compiler testing area is included in our dataset. We find that most papers are filtered out in this step because these papers are related to compiler verification or other software testing process. It is time-consuming and laborious work to exclude irrelevant papers. Nonetheless, we design and conduct such a concise search string to describe the compiler testing area and include many more papers that may be related to this area in the initial dataset. Manually checking on the papers can ensure that most papers related to compiler testing are included in our dataset, and filter out those papers that do not focus on compiler testing issues. Thus, only 51 papers are left in our dataset after this step. Last, we apply the same selection criteria to the reference lists of the selected 51 papers to find additional papers. Nine papers that are not retrieved by the search keywords are included. Finally, we obtain 60 papers related to compiler testing for the following procedures.

We design a data extraction form to collect needed information to support the bibliometric analysis and the collaboration analysis, as shown in Table 1. In addition to the bibliographic information of title, keywords, abstract, author(s), institution(s), country, and published year, the data form also includes the citation number of each paper which is collected from Google Scholar, the identified subject of compiler under test, the tools and the methods used for test case generation, and the types of compiler testing technologies.

Table 1 Extraction data item and description

Data item	Description
Title	Title of paper
Author	Authors'name of paper
Abstract	Abstract of paper
Keywords	Keywords presented on paper
Institution	Institution of author
Country	Country of author
Published year	Year that the paper was published
Citation	Citation number of paper
Subject	Types of compiler under test
Data generation	Tools/methods proposed to generate test case
Compiler testing technology	Types of testing method used

When we collect the bibliographic information from papers, we find that not all of the selected papers contain keywords due to the different formatting template of different journals/conferences. To accurately analyze the keywords, we furnish keywords information of these papers by extracting three keywords from the abstract information using the TextRank [42] algorithm, which is a graph-based ranking model for text processing, and has been successfully used in natural language applications for term identification [43, 44]. We select at least three keywords by the TextRank algorithm for each paper in the following analyses.

However, the items of subject, data generation, and compiler testing technology can not be directly extracted from papers. For these pieces of information, we employ three postgraduates of Dalian University of Technology to manually identify the relevant items. Each of them needs to scan each

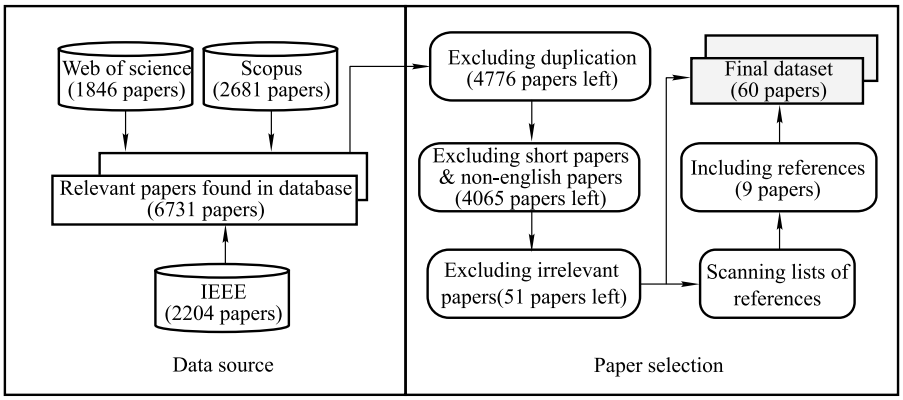


Fig. 3 The process of paper section

paper to answer the following questions:

- What types of compilers are tested in the paper?
- How the test cases are generated for compiler testing?
- Which testing technology is employed when testing compilers?

We adopt the most consistent answers for each question. If there are no consistent answers to a question, we invite another three postgraduates to answer the question until there is a consistent agreement.

All the needed information of selected papers in the data extraction form is constructed into our dataset. Subsequently, we conduct bibliometric analysis and collaboration analysis based on this dataset, and detail these analyses in the following subsections.

3.2 Bibliometric analysis

The bibliometric analysis consists of three modules, i.e., the productivity analysis, the impact analysis, and the content analysis. We show the details of each module in the following subsections.

3.2.1 Productivity analysis

The productivity analysis is mainly used to identify the most productive authors, institutions, countries, and popular topics in the compiler testing area. Thus, we calculate the number of papers for each author, institution, and country to identify the most productive ones. In order to avoid the ambiguity of the authors with the same name, we calculate the published number of each author with the institution when the paper is published. Once there are authors with the same name but different institutions, we check the homepage of authors to distinguish them. In addition, if the authors of a paper are from different institutions and counties, we calculate the distinct institution and country for once.

Then, we count the frequencies of keywords to identify the most popular topic and the trends of several popular topics. Notably, we delete the keywords "compiler testing" and "compiler bugs" when calculating the frequency of keywords, since the keywords are our search strings and the focus of this study.

3.2.2 Impact analysis

The impact analysis is used to identify the influential authors and papers in the compiler testing area. We detail the measurement of the impact of papers and authors as follows.

1) Impact of papers The motivation behind this indicator is that, the higher the citation number is, the higher impact of a paper receives. We use Google Scholar to find all papers' citation number before February 3rd, 2018. However, the newly published papers tend to have a smaller citation number compared with the previous ones. Therefore, we use normalized citation impact index (NCII) [45] which considers the impact of a publication's longevity to solve this issue. The score of NCII can be calculated as follows:

$$\text{NCII} = \frac{\textit{Total citation per referenced publication}}{\textit{Publication Longevity(inyears)}}. \quad (1)$$

Publication longevity indicates the number of years that a paper has been in print. With respect to this paper, the year 2018 is considered as the end point of the period.

2) Impact of authors We utilize individual contributions of papers to measure the impact of authors. Specifically, we employ adjusted citation score (ACS) [46], to calculate the individual contributions based on both papers' number of the author and the citation number of each paper.

Given a set of papers $P = \{p_1, \ldots, p_n\}$ and a set of published numbers $N = \{n_1, \ldots, n_n\}$, each paper p_i in P has been published by the corresponding n_i authors in N in our dataset. Then, the score of ACS is defined as follow:

$$ACS = \sum_{p \in P} \frac{\text{NCII}}{n}. \quad (2)$$

We modify the calculation of ACS, and replace the citation number of each paper with the score of NCII. Thus, Eq. (2) evaluates a paper's quality by the corresponding NCII value.

3.2.3 Content analysis

The content analysis is used to identify the frequently used compilers, popular test case generators, and testing technologies. Thus, we analyze the frequency of each compiler under test, the widely-used test case generator and the test suite, the compiler testing technology, and the approach based on the manual extraction data items in our dataset.

3.3 Collaboration analysis

The collaboration analysis is mainly used to reveal the cooperative relationships between authors and their interests. Thus, we generate three collaboration networks, i.e., the co-authorship network, the author co-keyword network, and the keyword co-occurrence network, to realize this analysis. We construct these networks because the collaborations between authors can be directly reflected in the co-authorship network,

the common interests among authors can be found in the author co-keyword network, and the core topics in compiler testing can be detected by similar keywords in the keyword co-occurrence network. We also employ community detecting algorithm [47] to find different communities in networks. In addition, all networks are visualized as undirect graphs, because the collaborations among authors and keywords can be undisputedly viewed as parallel.

3.3.1 Collaboration networks associations

The information of authors and keywords are needed to construct the networks. In the co-authorship network and the author co-keyword network, the nodes stand for authors, while the nodes in the keyword co-occurrence network stand for keywords. Specifically, we use association rule mining [48] to help mine useful collaboration associations.

As we are interested in constructing collaboration networks, we need to identify the frequent pairs of collaborations between authors and keywords. In the co-authorship network, a pair of authors is a frequent item if the proportion of the number of papers that are co-authored by this pair of authors is above the minimal support threshold t_s. Similarly, in the author co-keyword network, if the proportion of the number of papers that are organized using the same keywords by a pair of authors is above the minimal support threshold, we incorporate this pair of authors into frequent items. In the keyword co-occurrence network, a pair of keywords is a frequent item if the proportion of the number of papers that are organized with this pair of keywords is above the minimal support threshold. An association rule is generated from such pair if the confidence of this rule is above the minimal confidence threshold t_c. The confidence threshold is calculated as the proportion of the number of papers that contain the frequent pair of collaborations compared with the number of papers that contain only the first one in the frequent pair.

Given the mined association rules, we can construct three collaboration networks. Each network is an undirected graph $N = \{A/K, E, W\}$, where the node set A/K contains authors or keywords that appear in the association rules. The link set E contains undirect links that connect two authors or two keywords, and the weight set W represents the confident attribute indicating the strength of association rules.

3.3.2 Community detection

A network can consist of a large number of authors or keywords, as well as links between them. In graph theory, a node would be tightly linked with other relevant nodes, but loosely linked with irrelevant nodes. A set of highly correlated nodes is referred to as a community in the network. For example, in the author co-keyword network, authors with the same interests are most likely to be a community, because most of them focus on a specific topic in compiler testing. We use the Louvain method [47] implemented in the Gephi [49] tool to detect communities in the networks. The Louvain method partitions each network into a finite number of communities by using an iterative modularity maximization method, rather than requiring users to specify the number of communities. The modularity is defined as follow:

$$Q = \frac{1}{2m} \sum_{ij} \left[V_{ij} - \frac{d_i d_j}{2m} \right] \delta(c_i, c_j), \tag{3}$$

where the δ-function is 1 if nodes i and j belong to the same community, otherwise the δ-function is 0. Also, the V_{ij} is 1 if the two nodes i and j are linked, otherwise the V_{ij} is 0. m indicates the number of links in the network, and the d_i represents the degree number of the node i. Each node must be assigned to a specific community. Intuitively, the links in the same community will enhance the density of the network, and perform a positive effect to increase the modularity, whereas the links across different communities have a negative effect on modularity.

3.3.3 Visualizing the networks

We use the Gephi [49] tool to visualize the collaboration networks. Forceatlas2 layout [50] is used to achieve spatialization, because this layout is convenient to investigate different communities. Nodes and links in the same community are shown in the same color, whereas the nodes and links are shown with different or similar colors in different communities. The size of a node (author/keyword) represents the number of collaborations. The larger a node is, the more authors or keywords collaborate with the node. The thickness of links represents the strength of associations rules. The wider a link is, the more times that the two nodes collaborate with each other. However, the length of links bears no meaning in this paper due to the use of Forceatlas2 layout.

4 Results and analysis

In this section, we present the results of the analyses based on our framework using the constructing dataset. In particular, we investigate the following research questions:

RQ1 What are the influential authors, institutions, and the trends in the area of compiler testing?

RQ1.1 What are the productive authors, institutions or countries?

RQ1.2 What are the frequent keywords and the trends of popular topics?

RQ1.3 What are the influential authors and papers in the area of compiler testing?

RQ2 What are the research situations of compiler testing?

RQ2.1 What compilers are frequently tested?

RQ2.2 What test cases and testing technologies are employed when testing compilers?

RQ2.3 How to reduce the large test cases before reporting?

RQ3 What are the author communities and topic communities in the compiler testing area?

RQ3.1 What are the relationships among authors of compiler testing?

RQ3.2 What are the same interests of authors?

RQ3.3 What are the frequent co-occurrence keywords in the area of compiler testing?

We conduct the bibliometric analysis to help mine infrastructural information of compiler testing to address the former two main questions. Then, we conduct the collaboration analysis to explore the relationships among authors and keywords to address the last main question. In addition, we visualize the collaboration networks to characterize the collaborations more clearly.

4.1 Investigation to RQ1

We detect a large number of excellent authors and institutions that plays major roles in the development of the compiler testing area by conducting the productivity analysis and the impact analysis. In the following subsections, we only list some top-ranked results due to space restrictions.

4.1.1 RQ1.1 What are the productive authors, institutions or countries?

As for the statistical account of authors and institutions, we list the number of published papers for each author and institution in Fig. 4 and Fig. 5, respectively. The results show that several authors, such as Zhendong Su, Vu Le, and Chengnian Sun, have published more papers related to compiler testing. In addition, most productive authors have collaborations with others. For example, top three productive authors have co-published seven papers in our dataset. Other researchers also make many contributions to promote the development of compiler testing. When calculating the number of papers for each institution, we find that many universities have multiple campuses which are usually located in different areas, and have different research contributions. Thus, we distinguish each campus of a university, and find that the branch campus of University of California at Davis has published the most papers in the compiler testing area.

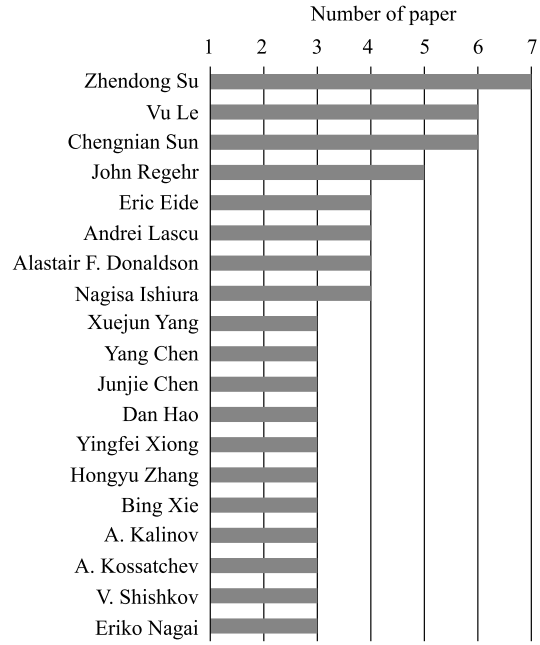


Fig. 4 The most productive authors

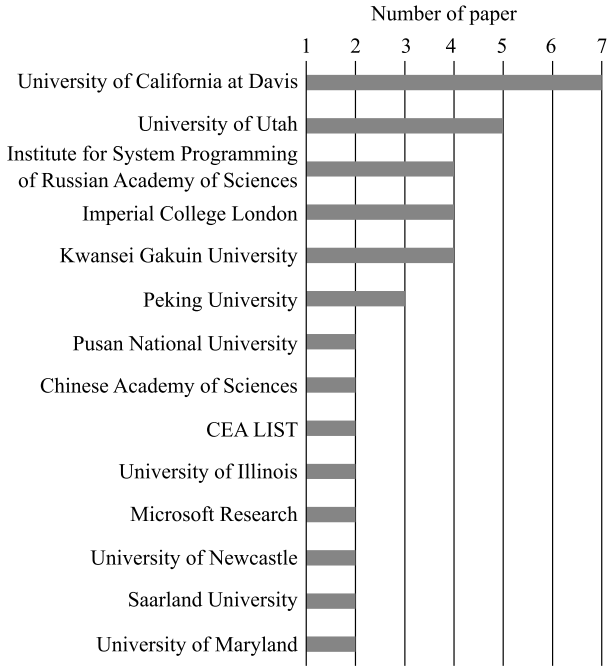


Fig. 5 The most productive institutions

In addition, we present the number of papers and the ratio of per country/region in Fig. 6. The results show that the

USA is the leading country with 30 published papers in our dataset, which is consistent with the results obtained in previous studies for ranking analyses of both paper quantity and quality [51]. We can also notice that Japan, the UK, China, and France are the most active countries, which indicates that the researchers in these countries tend to pay more attention to compiler testing.

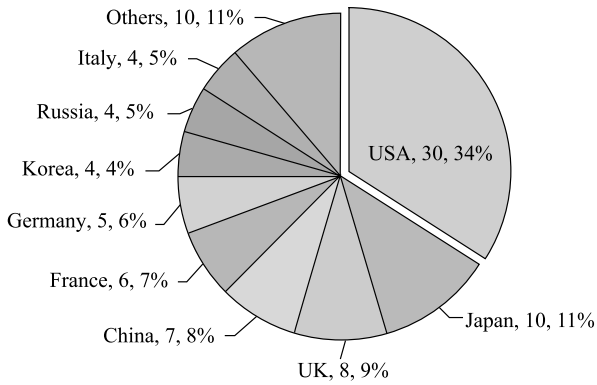


Fig. 6 The number of papers and ratio for countries/regions

4.1.2 RQ1.2 What are the frequent keywords and the trends of popular topics?

We also investigate the frequencies of keywords, and list the top-ranked keywords that occur more than three times in Fig. 7. The top three active keywords are "random testing", "test case generation", and "automated testing". The first keyword and the third keyword focus on mitigating the test oracle problem, and the second keyword aims to address the difficulty in test case generation.

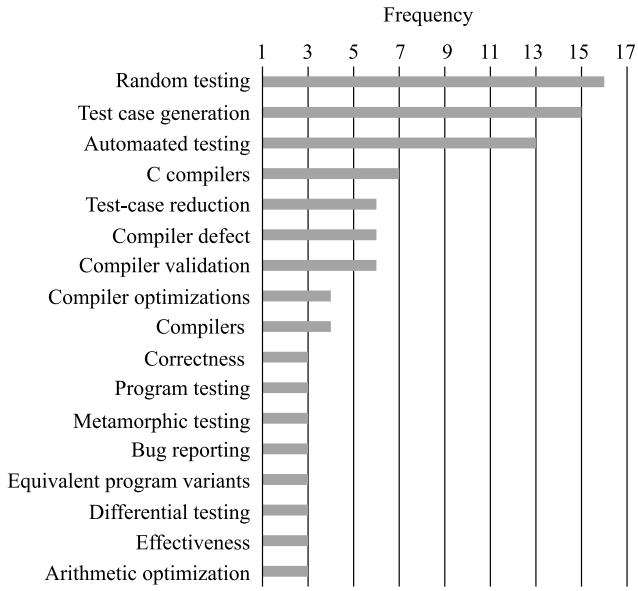


Fig. 7 The most frequent keywords

In addition, we analyze the trends of several keywords. As the papers in our dataset are published from 1976 to 2017, we split the papers into six periods. We accumulate the frequencies of top six keywords on each period to analyze the trends of them, as shown in Fig. 8. We can observe that the keyword "test case generation" has a sharp increase during 1997 and 2003, while presenting a smooth increase after 2003. Indeed, test case generation is a difficult task in compiler testing, and the existing generator tools are only prepared for several languages. In the future, there would be more generator tools to improve testing compilers that support various languages. Notably, the keyword "random testing" attracts much more attention after 2010, and becomes the most popular keyword in recent years. Simultaneously, the keyword "automatic testing" also shows a sharp increase during the past decade. The reason may be that as an automatic test case generation tool, CSmith, is proposed in 2011, enormous test cases are generated, making the random testing and automatic testing possible. Furthermore, as the compiler is generally of complex structure and the functionality of generating target machine code is the only concern, randomly automatic testing based on enormous test cases is critical in comprehensive testing compilers [52]. Other keywords also show a great increase after 2010, such as test-case reduction, which is an emerging topic in recent years.

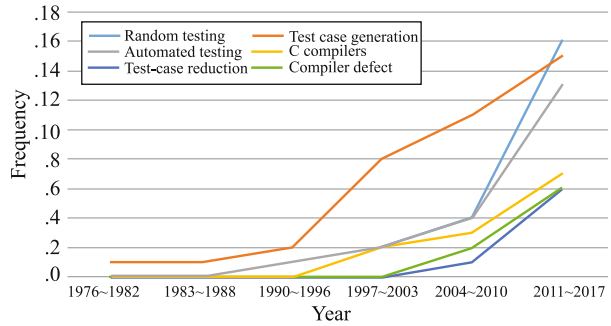


Fig. 8 The trends of top six keywords

4.1.3 RQ1.3 What are the influential authors and papers in the area of compiler testing?

As for the impact of authors, we calculate each ACS score, and list the top ten authors in Table 2. We can observe that John Regehr, Eric Eide, Yang Chen, Xuejun Yang, and Zhendong Su have higher ACS scores, which indicates that they are excellent researchers in the compiler testing area. Furthermore, we also investigate that most of these authors have published more than four papers in our dataset, such as Zhendong Su, Ve Le, John Regehr, and Eric Eide, which implies that authors with more published papers tend to have higher

impact in the compiler testing area.

Table 2 Top scores of ACS

No.	Author name	Score
1	John Regehr	23.00
2	Eric Eide	22.07
3	Yang Chen	18.47
4	Xuejun Yang	17.66
5	Zhendong Su	14.14
6	Vu Le	13.81
7	William M. McKeeman	10.40
8	Robert Mandl	9.97
9	Mehrdad Afshari	8.00
10	Chengnian Sun	6.14

As for the impact of papers, we calculate the NCII score of each paper, and list the top ten influential papers in Table 3. From the table, we can see that the most influential paper focuses on addressing the difficulty in test case generation, and creates a tool, Csmith, which can detect many unknown compiler bugs. We can also observe that most papers in top ten are published in the last decade, whereas only two papers are published in the period of eighties and nineties. As for the two early papers, one paper published in 1998 is the first time to propose differential testing technology to test C compilers, and emphasize the importance of avoiding undefined behaviors when generating C test programs, which attracts many following researches. The other paper published in 1985 designs an algebraic method for testing Ada compiler, which is widely-used to generate optimal test cases in software testing.

Answer to RQ1 By conducting the productivity analysis and the impact analysis, we find that the USA is the most influential country with a lot of excellent researchers and institutions in the compiler testing area. The keywords "random testing" and "automated testing" show a sharp increase in recent years and tend to be the most popular keywords from academia.

4.2 Investigation to RQ2

We detect the most frequently tested compilers, popular compiler testing technologies, and available tools by conducting the content analysis.

4.2.1 RQ2.1 What compilers are frequently tested?

We calculate the frequencies of tested compilers used by researchers, and list the number of papers of tested compilers in Fig. 9. Notably, most papers use various types of compilers that support the same language or one type of compiler with different optimization levels. The results show that C compilers are frequently tested by most papers, especially GCC and LLVM/Clang. In fact, GCC [3] is a compiler system supporting various languages and target architectures. LLVM [53] is another popular compiler infrastructure, and has drawn much attention from academia. Other compilers supporting different languages also attract researchers to test their correctness,

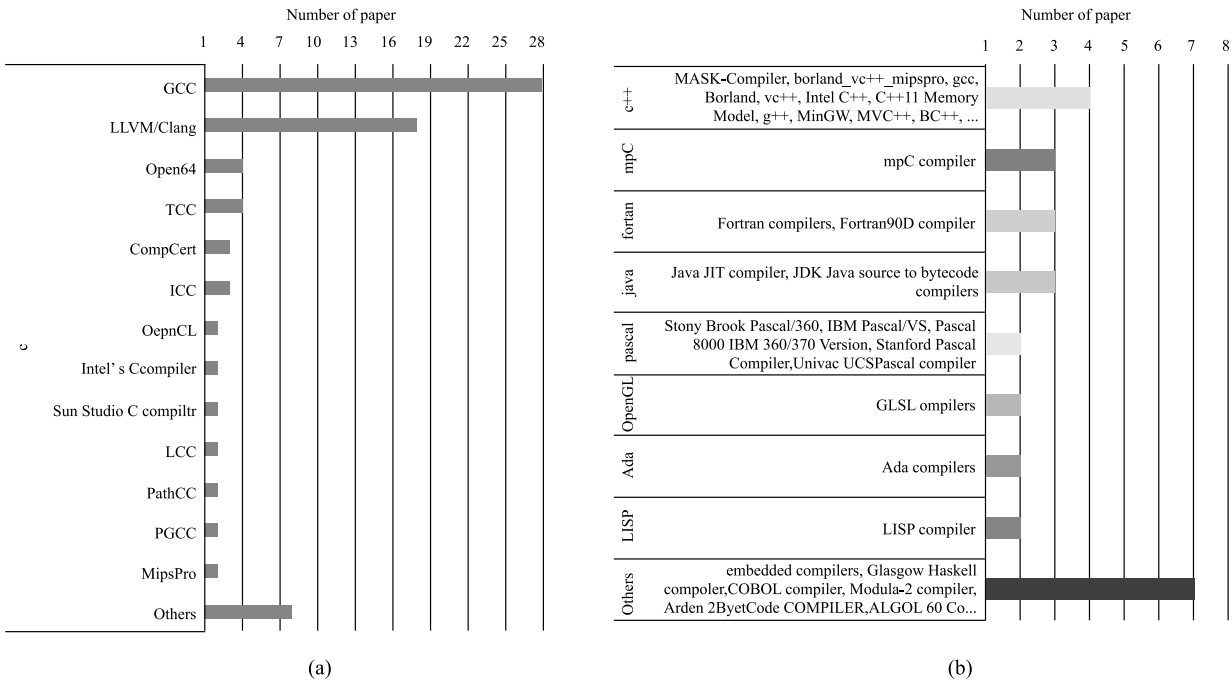


Fig. 9 Compiler under test supporting different languages. (a) Compiler under test supporting C language; (b) compiler under test supporting other languages

Table 3 The most influential papers

No.	Title	Year	Citation	NCII	Main contributions
1	Finding and understanding bugs in c compilers	2011	393	56.14	• A state-of-the-art test case generator named Csmith is developed, and many previous unknown compiler bugs are found • A qualitative and quantitative analysis is conducted to characterize the bugs
2	Compiler validation via equivalence modulo inputs	2014	96	24.00	• A novel testing technology of equivalence module input (EMI) is introduced • An instance of EMI named Orion is developed for testing C compilers • A large number of bugs in GCC and LLVM are reported by the evaluation of Orion
3	Test-case reduction for c compiler bugs	2012	97	16.17	• Three new, domain-specific test case reducers for C codes are proposed in a general framework • A crucial test-case validity problem is identified, and can be solved by various solutions • The best reducer is much more effective which produces test cases more than 25 times smaller than that produced by a delta debugger
4	Taming compiler fuzzers	2013	61	12.20	• The paper frames the fuzzer taming problem, which has not been addressed by researchers • The paper exploits the observation that automatic triaging of test cases and automatic test case reduction can be synergistic in accelerating compiler testing • The paper leverages diverse sources of information about bug-triggering test cases to rank test cases • The furthest point first (FPF) technology is both faster and more effective to cluster test cases than other clustering algorithms • Many bugs in a JavaScript engine and a C compiler are found during the fuzzing run
5	Differential testing for software	1998	208	10.40	• A new testing technology, differential testing, is proposed, and discovers new bugs in C compilers.
6	Many-core compiler fuzzing	2015	31	10.33	• The paper provides the evidence on the effectiveness of random differential testing and EMI testing in a new application domain • The paper proposes three novel methods for generating OpenCL kernels • An injection of dead-by-construction code enable EMI testing in the context of OpenCL • More than 50 OpenCL compiler bugs existing in commercial implementations are reported
7	Orthogonal latin squares: an application of experiment design to compiler testing	1985	327	9.91	• The paper proposes a new method for testing compilers, i.e., orthogonal latin squares, which can facilitate exhaustive testing at a fraction of the cost • The method is effective in designing some tests by using Ada Compiler Validation Capability test suites
8	Compiler testing via a theory of sound optimisations in the c11/c++11 memory model	2013	40	8.00	• A theory of sound optimizations in the C11/C++ memory model is proposed, which covers most optimizations in real compilers • A bug-hinting tool, cmmtest, is built based on the theory, and discovers some subtle concurrency bugs and unexpected bugs in the C11/C++ memory model
9	Testing an optimising compiler by generating random lambda terms	2011	53	7.57	• The paper provides a workable solution to generate random and type-correct lambda terms, and discovers many bugs in the Glasgow Haskell compiler
10	Volatiles are miscompiled, and what to do about it	2008	72	7.20	• The paper shows that C's volatile qualifier in compilers can produce incorrect object codes • The paper proposes a technique for generating C programs randomly • A new testing technique, access summary testing, is proposed, which is effective and automatical at detecting compiler bugs • The paper shows that the impact of compiler bugs can be mitigated by introducing small helper functions into a program • Several recommendations are provided for application developers and compiler developers

ranging from C++, Java to Pascal. As a result, the quality of compilers is critical for any language, and the compilers should be comprehensively tested.

4.2.2 RQ2.2 What test cases and testing technologies are employed when testing compilers?

In order to test compilers, test cases are needed as the inputs of compilers (see Section 3.2). To effectively generate abundant test cases that conform to language standards and specifications, many approaches and tools are proposed to generate random test cases without undefined behaviors. We identify each test case generator, and list the frequencies of these generators in Fig. 10. The results show that most test cases are generated based on the language grammar rules and

the coverage criteria. Several tools are frequently adopted by researchers, such as Csmith, Quest, CLsmith, Orange4, randprog, Epiphron, and JTT, whereas test suits are rarely employed because of the limitation of definite test cases. As for the automatic test case generators, Csmith is the most widely-used tool for C language test case generation, because Csmith covers a broad range of syntax of the C language, including arrays, structs, conditional statements, loop statements, and function calls, which is more expressive than other tools. In addition, several approaches can also generate abundant test cases. For example, Purdom' algorithm is a popular approach which generates test cases based on the language specifications, and has been extended by other test case generation approaches. Other approaches based on metamorphic testing and SPE can generate a set of semantic equivalence test cases as the inputs of compilers.

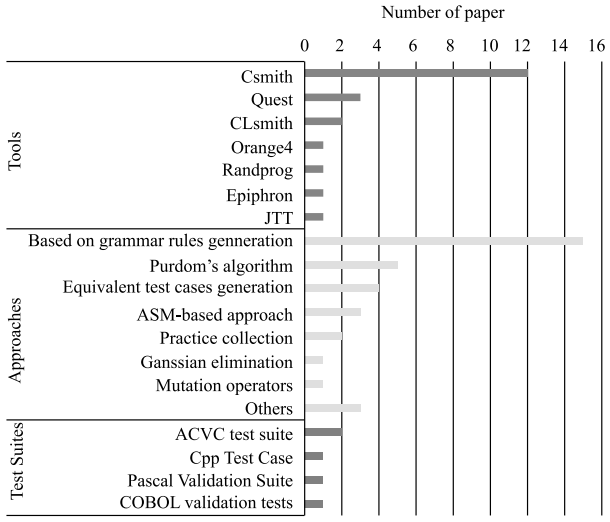


Fig. 10 Test case generation tool/approach

After the test cases are fed into compilers, we need to employ testing technology to test compilers based on these test cases (see Section 3.3). Different testing technologies are proposed to guarantee the quality of compilers as shown in Fig. 11. Random testing and differential testing are the two most frequently used technologies. Both of these two technologies can test compilers using randomly generated test cases as long as time allows. The view behind the differential testing is that if more than two compilers under the same test cases produce different results, there is a bug in at least one compiler. EMI derived from metamorphic testing attempts to construct equivalence-preservation relations to generate equivalent test cases for testing compilers. In addition, EMI is simple and widely applicable, which has been employed by many researchers. Mutation testing, as a trade off between the efficiency and the effectiveness in compiler testing, detects a mutant if errors manifest in a mutant, which is adopted by several researchers. Other testing approaches, such as Optimizer Testing Kit approach [54] and SPE approach, are also employed by researchers to test different compilers, and show their own effectiveness on bug detection.

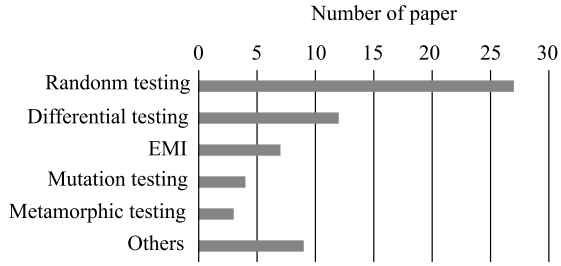


Fig. 11 Different compiler testing technologies

4.2.3 RQ2.3 How to reduce the large test cases before reporting?

Once a test case triggers a bug, the test case should be reduced before reporting, because large test cases are tedious and time-consuming for developers to find the root cause of the bug (see Section 3.4). Several automatic reducers are developed to help reduce large test cases, and ensure that the reduced test cases still trigger the same bug, and do not introduce new undefined behaviors. We identify each reducer employed by researchers, and list the frequencies of these reducers in Fig. 12. C-reduce is the most popular reducer due to the high efficiency and effectiveness on reducing test cases. Berkeley Delta and CL-Reduce are also adopted by researchers when there is a need to reduce the large size of test cases. Another reduction approach [39] employs top-down minimization and bottom-up minimization algorithms to reduce the arithmetic expressions to a small program.

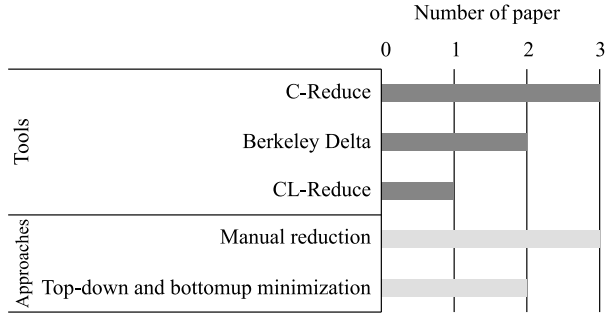


Fig. 12 Tools of test case reducer

Answer to RQ2 By conducting the content analysis, we find that C compilers are frequently tested by academia, and random testing is the most popular testing technology. In ad-

dition, several tools for test case generation and reduction are available for the public, such as Csmith and C-reduce. However, the number of papers implementing test case reduction is much smaller than that of test case generation, which also encourages researchers pay more attention on the large test case reduction, even for the real world projects.

4.3 Investigation to RQ3

We explore the relationships among authors and their interests by two collaboration networks using association rules, i.e., the co-authorship network and the author co-keyword network. In addition, we analyze the relationships between the frequent co-occurrence keywords by the keyword co-occurrence network. In the following subsections, we present and analyze the collaborations in these networks.

4.3.1 RQ3.1 What are the relationships among authors of compiler testing?

We present the collaboration relationships among authors in the co-authorship network. Actually, the number of authors in this network is affected by the minimal support t_s and the minimal confidence t_c. When the minimal t_s is set to 0.017, and the minimal t_c is set to 0, all the frequent pairs of authors will be included in the network. Thus, the co-authorship network has the maximum number of authors in the compiler testing area.

The co-authorship network is shown in Fig. 13 which contains 119 authors and 229 links, and includes 27 authorship communities with the modularity of 0.893. In Fig. 13, authors in compiler testing distribute in several scattered communities, which only 32 authors (27% rate) collaborate with more than five authors, and only one author (0.8% rate) collaborates with more than ten authors. The strength of collaborations is in a small rang from one to six cooperative times, and only five pairs of authors (2.18% rate) collaborate with each other more than three times. These communities are isolated from each other, among which seven communities follow an edge structure, six communities follow a triangle structure, four communities follow a quadrilateral structure, while the other ten communities follow a complex network structure. In the following discussions, we only discuss the ten complex communities, and use the high degree author or the productive author to represent a community, such as the John Regehr community and the Junjie Chen community.

In Fig. 13, the John Regehr community is the most complex community with 14 collaborators and 51 links. In this community, John Regehr is the central author, and has other 13 collaborators, especially collaborating with Eric Eide for four times. John Regehr is also a productive author in Fig. 4,

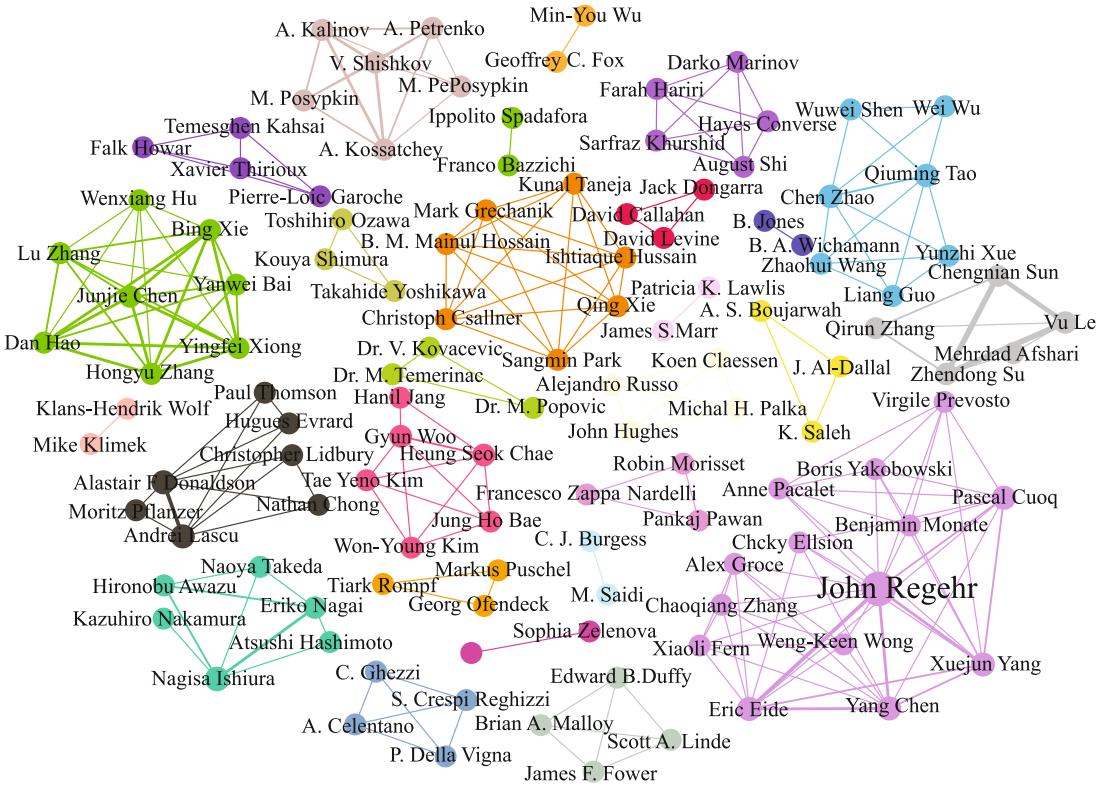


Fig. 13 Co-authorship network

and receives the most highest ACS score in Table 2. Other productive authors also appear in this community, such as Yang Chen and Xuejun Yang. All of these authors have broad collaborations that form the biggest community in the compiler testing area. In the Junjie Chen community and the Ishtiaque Hussain community, there are eight and seven collaborators respectively. Each pair of the collaborators in these two communities has co-authored papers in our dataset. Specifically, there are three co-authored papers related to compiler testing in the Junjie Chen community as shown in Fig. 4.

The other seven complex communities have more than five collaborators, and surround with several productive authors, such as Zhendong Su and Alastair F. Donaldson. Although there are only five authors in the Zhendong Su community, all the authors dominate the state-of-the-art technologies on compiler testing. Another two communities, i.e., the Alastair F. Donaldson community and the Nagisa Ishiura community, have several strong associations among collaborators, while other four complex communities have more collaborators but weak associations.

We also investigate the determining factors of the co-authorship phenomenon, and the impact of papers affected by the collaborations. In our dataset, the authors of more than half of papers are from the same institution, and of more than two thirds papers are from the same country. For examples, all the authors in the Zhendong Su community are from the University of California at Davis, and all the authors in the Junjie Chen community are from China. Furthermore, collaborations can increase the number of papers, as well as the accepted rate for publication in a conference or journal. However, the quality of co-authored work is the most critical factor on the acception for a top conference or journal, which can greatly improve the influence of a paper. In fact, collaborations can certainly improve the quality of work, especially collaborating with some productive authors, such as the most influential papers listed in Table 3, which has received much more citation numbers from academia.

4.3.2 RQ3.2 What are the same interests of authors?

In this subsection, we analyze the interests among authors using the author co-keyword network. Similarly, given the minimal support t_s of 0.03, if the minimal confidence t_c is set to 0.09, authors that published more than two papers using the same keywords more than three times are included in this network. We set this pair of parameters in association rules because it can significantly detect the same interests among productive authors, and present a clear topological structure in the network.

The author co-keyword network is shown in Fig. 14. We can see that there are six communities clustered by 29 authors and 65 links. Each community is composed of authors with the same topic because the authors in a community tend to use the same keywords. Thus, different communities share different topics in the compiler testing area. As the same in co-authorship network, we use the high degree author or the productive author to represent a community, such as the Alastair F. Donaldson community and the Junjie Chen community.

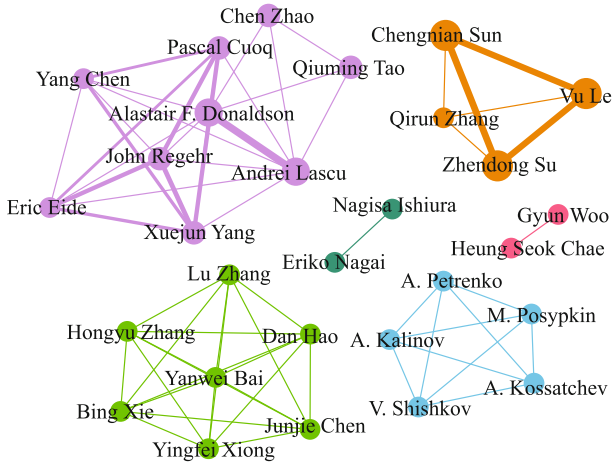


Fig. 14 Author co-keyword network

In Fig. 14, six communities are isolated from others. Among these communities, the largest community is dominated by Alastair F. Donaldson, who is interested in graphics shader compilers and many-core compiler testing, and focuses on CLsmith and CL-reduce tools for test case generation and reduction. Furthermore, this community is composed of three collaboration communities in co-authorship network, namely the John Regehr community, the Alastair F. Donaldson community and the Qiuming Tao community, as shown in Fig. 13. The Zhendong Su community aims at issues of test case generation and compiler testing technology, such as SPE and EMI.

The other three isolated communities have the same collaboration communities as Fig. 13 shows. The Junjie Chen community aims to prioritize test cases for compilers to accelerate the process of compiler testing [37,55]. The A. Kalinov community focuses on test case generation for mpC compiler [29], and the Heung Seok Chae community is interested in test case reduction for retargeted compilers [27,56].

4.3.3 RQ3.3 What are the frequent co-occurrence keywords in the area of compiler testing?

In this subsection, we present the major topics and the links

between keywords by the keyword co-occurrence network. We analyze the structure of this network mined at the minimal support t_s of 0.033. When the minimal confidence t_c is set to 0, all the keywords that occurred more than two times are included in this network. We select these parameters due to two reasons. First, 33.64% frequent keywords can be included in this network when the minimal set is 0.033. Second, we can discover a clear topological structure among these frequent keywords.

The keyword co-occurrence network is shown in Fig. 15 which consists of 37 keywords and 117 links, and forms five communities with the modularity of 0.248. We use the high degree node or the major topic to define a community. Furthermore, to avoid the ambiguity with nodes in a community, we use the first-word-capitalized name to refer to a community, such as the Compiler Test community.

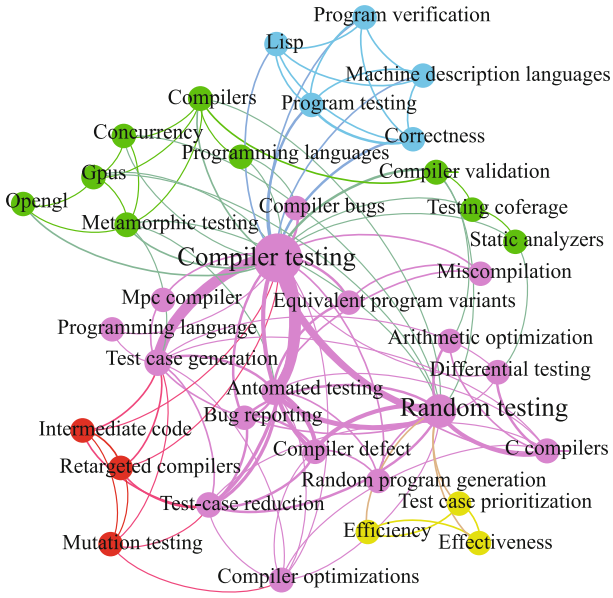


Fig. 15 Author co-keyword network

In Fig. 15, most communities have complex links with each other. We can see that a central keyword "compiler testing" is linked with most keywords in these communities. In addition, two keywords "random testing" and "test case generation" have strong associations with the central keyword, indicating that these two keywords are the most important topics related to compiler testing, and attract more researchers to focus on. Simultaneously, these three core keywords dominate the largest community in this network which we define as the Compiler Test community. In this community, there are 17 keywords, including the issues and the solutions related to compiler testing, such as "test-case reduction", "equivalent program variants", "random program generation", and "differential testing", which defines a fine-grained category of compiler testing. Another three keywords, namely "efficiency", "effectiveness", and "test case prioritization", also surround with a high degree keyword "random testing", which form a community aimed at improving the performance of compiler testing technologies.

The Compiler Validation community is also a relatively larger community that contains nine keywords which focuses on test case generation for many core compilers. This community surrounds with keywords "gpus", "opengl", "concurrency", "testing coverage", "static analyzers", "compiler validation", "programming languages", "compilers", and "metamorphic testing". However, two other communities are relatively smaller. The keywords in these two communities are linked with each other which referred as the Program Verification community and the Retargeted Compiler community, respectively.

Answer to RQ3 With the analysis of the co-authorship network and the author co-keyword network, we find that the co-authorship in the compiler testing area distributes in several scattered communities. Authors in the same institution and the same country tend to collaborate with each other. In addition, most productive authors have broad interests, and the collaborations with the productive authors can improve the influences of papers to a certain extent. By constructing the keyword co-occurrence network, we find that the test case generation and the test oracle problem are the two most critical issues in compiler testing, which surround with abundant relevant keywords.

5 Related work

The most relevant work is literature analysis. In this section, the majority of related work can be classified into two aspects, i.e., the bibliometric analysis and the collaboration analysis.

5.1 Bibliometric analysis

A large number of bibliometric studies have been published in software engineering. Wohlin et al. [57–60] analyzed the highly cited papers in software engineering published from 1999 to 2002. Wong et al. [61–63] identified top-15 researchers and institutions for two five-year periods between 2008 and 2011. The rankings were based on the number of published papers from seven leading software engineering journals.

Focusing on the sub-areas of software engineering, Feitas

and Souza [64] presented a bibliometric analysis for ten years of search-based software engineering that covered 740 papers from 2001 to 2010. Jiang et al. [65] constructed a publication analysis framework to present some important domain knowledge for mining software repositories. Some recent systematic mapping studies also included bibliometric analysis of sub-areas of software engineering, e.g., Web application testing [40].

In previous work, Garousi and Ruhe [66] conducted the first quantitative bibliometric analysis in total about 60% of the software engineering literature, and reported interesting findings, such as the USA is the clear leader, but the contributions to software engineering by the American researchers have decreased from 71.43% (in 1980) to 14.90% (in 2008). More recently, Garousi and Fernandes [67] utilized automated topic analysis to characterize and understand massive software engineering literature.

5.2 Collaboration analysis

The co-authorship network aims to find the cooperative relationship among authors. Velden et al. [68] studied patterns of collaboration in the co-authorship networks with the data obtained from Web of Science. They identified two types of coauthor-linking patterns between authorship communities with the name disambiguation. But they distorted the topological structure of the co-authorship networks in some cases because a small set of common surnames are widely used in some East Asian countries. Madaan and Jolad [69] found interesting features in the co-authorship network, such as the collaborations between researchers is increasing over time, and few researchers published a large number of papers in DBLP Computer Science Bibliographic database.

Su and Lee [51] created a three-dimensional research, focusing on a parallel network, i.e., the keyword co-occurrence network, and a two-dimensional knowledge map to visualize the knowledge structure using the data of journal papers.

The difference between our work and previous work is that we employ both bibliometric and collaboration analyses for compiler testing literature analysis. In the bibliometric analysis, we not only distinguish the authors' names with their institutions to avoid ambiguities, but also combine the ACS score and the NCII score to measure the impact of authors. In the collaboration analysis, we first incorporate the social network and the data mining technique to construct the co-authorship network, the author co-keyword network, and the keyword co-occurrence network to help mine useful collaborations.

6 Conclusion & future work

In this study, we present a literature analysis framework, to comprehensively characterize and understand the compiler testing area. We illustrate how each component works in the framework and obtain some useful information after conducting each component. The major contributions of this paper include two aspects. In the aspect of bibliometics analysis, we find that the USA dominates the area of compiler testing, having a large number of influential researchers, such as Zhendong Su, Vu Le, and Chengnian Sun. The keyword "random testing" is the most frequently used keyword by researchers, and C compilers are the most frequently tested compilers. In the aspect of collaboration analysis, we construct three collaboration networks, and find that collaborations with productive researchers can improve both the accepted rate and the quality of papers in the co-authorship network. In addition, we detect several researchers with the same interests in the author co-keyword network, and some fine-grained categories of compiler testing in the keyword co-occurrence network.

Although the previous work has proposed various solutions to the issues existing in the compiler testing area, there still remain several interesting challenges that need to be addressed in the future, such as using the real-world projects to test compilers, reducing test cases for multiple files, improving both the effectiveness and efficiency of compiler testing technologies, etc. In the future, we will focus on these challenges to improve test compilers and hope more researchers devote to compiler testing to boom this area.

Acknowledgements We would like to thank all the participants for the comments on improving this paper. This research was supported by the National Key Research and Development Program of China (2018YFB1003900), the National Natural Science Foundation of China (Grant Nos. 61722202, 61772107 and 61572097), and the Fundamental Research Funds for the Central Universities (DUT18JC08).

References

1. Howard M. A process for performing security code reviews. IEEE Security and Privacy, 2006, 4(4): 74–79
2. Pearse T, Oman P. Maintainability measurements on industrial source code maintenance activities. In: Proceedings of the International Conference on Software Maintenance. 1995, 295–303
3. Sun C, Le V, Zhang Q, Su Z. Toward understanding compiler bugs in GCC and LLVM. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016, 294–305
4. Sun C, Le V, Su Z. Finding and analyzing compiler warning defects. In: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering. 2016, 203–213

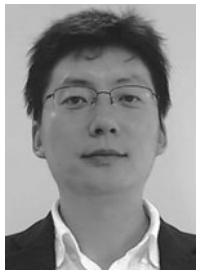
5. Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014, 216–226
6. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. 2011, 283–294
7. Chen J, Hu W, Hao D, Xiong Y, Zhang H, Lu Z, Xie B. An empirical comparison of compiler testing techniques. In: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering. 2016, 180–190
8. Lidbury C, Lascu A, Chong N, Donaldson A F. Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2015, 65–76
9. Sheridan F. Practical testing of a C99 compiler using output comparison. Software: Practice and Experience, 2007, 37(14): 1475–1488
10. Nagai E, Hashimoto A, Ishiura N. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. IPSJ Transactions on System LSI Design Methodology, 2014, 7(4): 91–100
11. Chen Y, Groce A, Zhang C, Wong W K, Fern X, Eide E, Regehr J. Taming compiler fuzzers. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2013, 197–208
12. Regehr J, Chen Y, Cuoq P, Eide E, Ellison C, Yang X. Test-case reduction for C compiler bugs. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. 2012, 335–346
13. Lindig C. Find a compiler bug in 5 minutes. British Journal of Ophthalmology, 2005, 79(4): 387–396
14. Lindig C. Random testing of C calling conventions. In: Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging. 2005, 3–12
15. Eide E, Regehr J. Volatiles are miscompiled, and what to do about it. In: Proceedings of the 8th ACM International Conference on Embedded Software. 2008, 255–264
16. Zhao C, Xue Y, Tao Q, Guo L, Wang Z. Automated test program generation for an industrial optimizing compiler. In: Proceedings of ICSE Workshop on Automation of Software Test. 2009, 36–43
17. McKeeman W M. Differential testing for software. Digital Technical Journal, 1998, 10(1): 100–107
18. Le V, Sun C, Su Z. Randomized stress-testing of link-time optimizers. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015, 327–337
19. Hariri F, Shi A, Converse H, Khurshid S, Marinov D. Evaluating the effects of compiler optimizations on mutation testing at the compiler ir level. In: Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering. 2016, 105–115
20. Tao Q, Wu W, Zhao C, Shen W. An automatic testing approach for compiler based on metamorphic testing technique. In: Proceedings of the 17th Asia Pacific Software Engineering Conference. 2010, 270–279
21. Donaldson A F, Lascu A. Metamorphic testing for (graphics) compilers. In: Proceedings of the 1st International Workshop on Metamorphic Testing. 2016, 44–47
22. Pflanzer M, Donaldson A F, Lascu A. Automatic test case reduction for opencl. In: Proceedings of the 4th International Workshop on OpenCL. 2016, 1–12
23. Ren Z, Jiang H, Xuan J, Yang Z. Automated localization for unreproducible builds. In: Proceedings of the 40th International Conference on Software Engineering. 2018, 71–81
24. Jiang H, Li X, Yang Z, Xuan J. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In: Proceedings of the 39th International Conference on Software Engineering. 2017, 712–723
25. Celentano A, Reghizzi S C, Vigna P D, Ghezzi C, Granata G, Savoretti F. Compiler testing using a sentence generator. Software: Practice and Experience, 1980, 10(11): 897–918
26. Boujarwah A S, Saleh K, Al-Dallal J. Testing syntax and semantic coverage of Java language compilers. Information and Software Technology, 1999, 41(1): 15–28
27. Chae H S, Woo G, Kim T Y, Bae J H, Kim W Y. An automated approach to reducing test suites for testing retargeted C compilers for embedded systems. Journal of Systems and Software, 2011, 84(12): 2053–2064
28. Wu M Y, Fox G C. A test suite approach for Fortran90D compilers on MIMD distributed memory parallel computers. In: Proceedings of Scalable High Performance Computing Conference. 1992, 393–400
29. Kalinov A, Kossatchev A, Posypkin M, Shishkov V. Using ASM specification for automatic test suite generation for mpC parallel programming language compiler. In: Proceedings of the 4th International Workshop on Action Semantic. 2002, 99–109
30. Zhang Q, Sun C, Su Z. Skeletal program enumeration for rigorous compiler testing. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017, 347–361
31. Barr E T, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: a survey. IEEE Transactions on Software Engineering, 2015, 41(5): 507–525
32. Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009, 52(7): 107–115
33. Kong W, Liu L, Ando T, Yatsu H, Hisazumi K, Fukuda A. Facilitating multicore bounded model checking with stateless explicit-state exploration. The Computer Journal, 2014, 58(11): 2824–2840
34. Le V, Sun C, Su Z. 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. 2015, 50(10): 386–399
35. Sun C, Le V, Su Z. Finding compiler bugs via live code mutation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications. 2016, 849–863
36. Mei H, Hao D, Zhang L, Zhang L, Zhou J, Rothermel G. A static approach to prioritizing junit test cases. IEEE Transactions on Software Engineering, 2012, 38(6): 1258–1275
37. Chen J, Bai Y, Hao D, Xiong Y, Zhang H, Xie B. Learning to prioritize test programs for compiler testing. In: Proceedings of the 39th International Conference on Software Engineering. 2017, 700–711
38. Li X, Jiang H, Liu D, Ren Z, Li G. Unsupervised deep bug report summarization. In: Proceedings of the 26th International Conference on Program Comprehension. 2018, 144–155
39. Nagai E, Awazu H, Ishiura N, Takeda N. Random testing of C compil-

ers targeting arithmetic optimization. In: Proceedings of the Workshop on Synthesis and System Integration of Mixed Information Technologies. 2012, 48–53

40. Garousi V, Mesbah A, Betin-Can A, Mirshokraie S. A systematic mapping study of Web application testing. Information and Software Technology, 2013, 55(8): 1374–1396
41. Kanewala U, Bieman J M. Testing scientific software: a systematic literature review. Information and Software Technology, 2014, 56(10): 1219–1232
42. Mihalcea R, Tarau P. Textrank: bringing order into text. In: Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing. 2004, 404–411
43. Balcerzak B, Jaworski W, Wierzbicki A. Application of TextRank algorithm for credibility assessment. In: Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). 2014, 451–454
44. Rahman M M, Roy C K. TextRank based search term identification for software change tasks. In: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution and Reengineering. 2015, 540–544
45. Holsapple C W, Johnson L E, Manakyan H, Tanner J. Business computing research journals: a normalized citation analysis. Journal of Management Information Systems, 1994, 11(1): 131–140
46. Mcclure C R. Foundations of library and information science. Journal of Academic Librarianship, 1998, 24(6): 491–492
47. Blondel V D, Guillaume J L, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008, 2008(10): 10008–10020
48. Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proceedings of the 20th International Conference on Very Large Data Bases. 1994, 487–499
49. Bastian M, Heymann S, Jacomy M. Gephi: an open source software for exploring and manipulating networks. In: Proceedings of International Conference on Weblogs and Social Media. 2009, 361–362
50. Jacomy M, Venturini T, Heymann S, Bastian M. ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. Public Library of Science One, 2014, 9(6): e98679
51. Su H N, Lee P C. Mapping knowledge structure by keyword cooccurrence: a first look at journal papers in technology foresight. Scientometrics, 2010, 85(1): 65–79
52. Mei H, Zhang L. Can big data bring a breakthrough for software automation. Science China (Information Sciences), 2018, 61(5): 056101
53. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. 2004, 75–86
54. Zelenov S, Zelenova S. Model-based testing of optimizing compilers. In: Proceedings of the International Conference on Testing of Software and Communicating Systems. 2007, 365–377
55. Chen J, Bai Y, Hao D, Xiong Y, Zhang H, Zhang L, Xie B. Test case prioritization for compilers: a text-vector based approach. In: Proceedings of 2016 IEEE International Conference on Software Testing, Verification and Validation. 2016, 266–277
56. Woo G, Chae H S, Jang H. An intermediate representation approach to reducing test suites for retargeted compilers. In: Proceedings of the International Conference on Reliable Software Technologies. 2007, 100–113
57. Wohlin C. An analysis of the most cited articles in software engineering journals — 1999. Information and Software Technology, 2005, 47(15): 957–964
58. Wohlin C. An analysis of the most cited articles in software engineering journals — 2000. Information and Software Technology, 2007, 49(1): 2–11
59. Wohlin C. An analysis of the most cited articles in software engineering journals — 2001. Information and Software Technology, 2008, 50(1–2): 3–9
60. Wohlin C. An analysis of the most cited articles in software engineering journals — 2002. Information and Software Technology, 2009, 50(1): 3–6
61. Wong W E, Tse T H, Glass R L, Basili V R, Chen T Y. An assessment of systems and software engineering scholars and institutions (2001—2005). Journal of Systems and Software, 2008, 81(6): 1059–1062
62. Wong W E, Tse T H, Glass R L, Basili V R, Chen T Y. An assessment of systems and software engineering scholars and institutions (2002—2006). Journal of Systems and Software, 2009, 82(8): 1370–1373
63. Wong W E, Tse T H, Glass R L, Basili V R, Chen T Y. An assessment of systems and software engineering scholars and institutions (2003—2007 and 2004—2008). Journal of Systems and Software, 2011, 84(1): 162–168
64. Freitas F G, Souza J T. Ten years of search based software engineering: a bibliometric analysis. In: Proceedings of the International Symposium on Search Based Software Engineering. 2011, 18–32
65. Jiang H, Chen X, Zhang J, Han X, Xu X. Mining software repositories: contributors and hot topics. Journal of Computer Research and Development, 2016, 53(12): 2768–2782
66. Garousi V, Ruhe G. A bibliometric/geographic assessment of 40 years of software engineering research (1969—2009). International Journal of Software Engineering and Knowledge Engineering, 2013, 23(9): 1343–1366
67. Garousi V, Fernandes J M. Highly-cited papers in software engineering: the top-100. Information and Software Technology, 2016, 71(3): 108–128
68. Velden T, Haque A, Lagoze C. A new approach to analyzing patterns of collaboration in co-authorship networks: mesoscopic analysis and interpretation. Scientometrics, 2010, 85(1): 219–242
69. Madaan G, Jolad S. Evolution of scientific collaboration networks. In: Proceedings of 2014 IEEE International Conference on Big Data. 2014, 7–13

Yixuan Tang received the BSc degree in computer science and technology from Liaoning University, China in 2015. She is currently a PhD candidate in Dalian University of Technology, China. Her current research interests include software data analytics and mining software repositories.

Zhilei Ren received the BSc degree in software engineering and the PhD degree in computational mathematics from Dalian University of Technology, China in 2007 and 2013, respectively. He is currently an associate professor with Dalian University of Technology. His current research interests include evolutionary computation, automatic algorithm configuration, and mining software repositories.



Weiqiang Kong received the PhD degree in information science from Japan Advanced Institute of Science and Technology in 2006. He is currently a professor with Dalian University of Technology, China. His current research interests include software engineering and formal methods (formal verification).

He Jiang is an awardee of the NSFC Excellent Young Scholars Program in 2017. He is currently a professor with Dalian University of Technology and an adjunct professor with Beijing Institute of Technology, China. His current research interests include search-based software engineering and mining software repositories. He has published over 60 referred papers on journals and international conferences, including IEEE Trans. Software Engineering, IEEE Trans. Knowledge and Data Engineering, ICSE, SANER, etc., supported by the Program for New Century Excellent Talents in University and the National Science Fund for Excellent Young Scholars. In addition, he serves as the guest editors of some journals and magazines, including IEEE Computational Intelligence, Journal of Computer Science and Technology, Frontiers of Computer Science, etc.