Defect Categorization in Compilers: A Multi-vocal Literature Review

AKOND RAHMAN, Auburn University, USA

DIBYENDU BRINTO BOSE, Virginia Tech, USA

FARHAT LAMIA BARSHA, Tennessee Technological University, USA

RAHUL PANDITA, Github, USA

Context: Compilers are the fundamental tools for software development. Thus, compiler defects can disrupt development productivity and propagate errors into developer-written software source code. Categorizing defects in compilers can inform practitioners and researchers about the existing defects in compilers and techniques that can be used to identify defects systematically.

Objective: *The goal of this paper is to help researchers understand the nature of defects in compilers by conducting a review of Internet artifacts and peer-reviewed publications that study defect characteristics of compilers.*

Methodology: We conduct a multi-vocal literature review (MLR) with 26 publications and 32 Internet artifacts to characterize compiler defects.

Results: From our MLR, we identify 13 categories of defects, amongst which optimization defects have been the most reported defects in our artifacts publications. We observed 15 defect identification techniques tailored for compilers and no single technique identifying all observed defect categories.

Conclusion: Our MLR lays the groundwork for practitioners and researchers to identify defects in compilers systematically.

CCS Concepts: • **Software and its Engineering → Compilers**.

Additional Key Words and Phrases: compiler, defect, internet artifact, review

ACM Reference Format:

Akond Rahman, Dibyendu Brinto Bose, Farhat Lamia Barsha, and Rahul Pandita. 2022. Defect Categorization in Compilers: A Multi-vocal Literature Review. 1, 1 (September 2022), 37 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' addresses: Akond Rahman, Auburn University, Auburn, AL, USA, akond@auburn.edu; Dibyendu Brinto Bose, Virginia Tech, Blacksburg, VA, USA, brintodibyendu@vt.edu; Farhat Lamia Barsha, Tennessee Technological University, Cookeville, TN, USA, fbarsha42@tntech.edu; Rahul Pandita, Github, Denver, CO, USA, rahulpandita@github.com.

Manuscript submitted to ACM

1 INTRODUCTION

According to the State of the Developer 2021 report, 26.8 million professionals worldwide are software developers [15]. These software developers rely on compilers to develop computer programs. Compilers are software systems that convert a computer program written in one programming language(typically higher-level) to low-level instructions, such as machine code. While performing this translation, the compilers also ensure computer programs that are being compiled abide by the syntactic and semantic rules of the programming language, and later the translated machine code is semantically equivalent to the compiled computer program. In this manner, compilers help software developers ensure the program performs desirable when executed, which allows developers to become productive. According to Sun et al. [56], "*Compilers are among the most important, widely-used system software, on which all programs depend for compilation*". A compiler is considered an important part of the software supply chain [17].

Despite being a 'fundamental programming tool' in software development [65], compilers themselves are software programs and thus prone to defects that can have severe consequences for software development. A compiler defect can propagate into all computer programs that are compiled by the defective compiler [51]. Defects in compilers have also been attributed to catastrophic consequences in safety-critical domains [56]. These defects are prevalent in well-known compilers: according to Marcozzi et al. [35], multiple defects in the Clang/LLVM and GCC compilers are fixed each month. Defects in compilers can be consequential for software developers with respect to productivity. For example, one software developer was stuck for five days due to a compiler defect [64].

The prevalence and consequences of defects in compilers necessitate systematic endeavors from the practitioner and research community to identify latent defects in compilers. These endeavors can be informed by a review of existing literature related to the defect characteristics of compilers. Such a review can systematically categorize the defects in compilers and also map techniques that are used to identify each of the defect categories.

As compilers play a pivotal role in professional software development that involves software practitioners, we want to get a practitioner's perspective of reported compiler defects. In that manner, we cannot only synthesize compiler defects reported by academics but also synthesize the defects reported by practitioners. Such analysis can aid the entire software engineering community by finding the commonalities and differences in the analyses and derive insightful recommendations. According to Garousi et al. [22] review of Internet artifacts can "enable a rigorous identification of emerging research topics in SE as many research topics already stem from software industry". Internet artifacts was used to curate best practices for continuous deployment [46], devops security [59], securing Kubernetes installations [53], and managing secrets with secret management tools [41].

Multi-vocal literature review incorporates both: review of Internet artifacts and a review of peer-reviewed publications [23]. Accordingly, we use a multi-vocal literature review (MLR) so that we can capture insights from academics as from software practitioners.

Objective: *The goal of this paper is to help researchers understand the nature of defects in compilers by conducting a review of Internet artifacts and peer-reviewed publications that study defect characteristics of compilers.*

To achieve our goal, in this work, we answer the following research questions:

- **RQ1**: *Which compilers have been studied in Internet artifacts and peer-reviewed publications that have investigated defects in compilers?*

- **RQ2**: *What categories of defects have been reported in Internet artifacts and peer-reviewed publications that have investigated defects in compilers?*

- **RQ3**: *What techniques have been reported in Internet artifacts and peer-reviewed publications to identify defects in compilers?*

We conduct our MLR with 32 Internet artifacts and 26 publications. We have conducted an MLR that requires analysis of two kinds of resources: Internet artifacts that are not peer-reviewed and publications that are peer-reviewed. Without the analysis of Internet artifacts, an MLR will be deemed incomplete and incorrect. Our use of Internet artifacts makes the MLR complete and also is useful to generate interesting insights. Using Kithchenham et al. [27] and Gharousi et al. [22]'s guidelines, respectively, we perform a quality evaluation of the 26 publications and 32 Internet artifacts. We apply a qualitative analysis technique called open coding [48] to determine defect categories reported in Internet artifacts and peer-reviewed publications. We have added the results from our multi-vocal literature reviews as a PDF in our replication package [40].

For the scope of our study, we define a compiler as a special type of software that takes source code as input and provides machine code or binary executables as input. This software category can support multiple languages and have multiple compilation engines to support each of these languages. Furthermore, based on our definition, this type of software can provide interfaces to develop even more compilation units and provide rich software development experience so that along with generating machine code or binary executables, users can perform testing, linting, and version control.

Compilers are used by a wide range of users, including academics who conduct scientific research and practitioners who use compilers to develop software. As such, the experiences of compiler usage as manifested in terms of defects needs to be included while conducting a review of compiler defect categories. Accordingly, we select an MLR instead of a systematic literature review (SLR) so that we can gain the perspectives of both academics and practitioners when it comes to defect categories for compilers. An MLR consists of reviewing two types of artifacts: academic publications that are peer-reviewed and artifacts that are practitioner-reported and not peer-reviewed by the research community [22]. The goal of using an MLR is to capture evidence from both worlds: the academic world and the practitioner world. Many practitioners tend not to participate in academic conferences, where academics come and present their findings. Instead, practitioners participate in practitioner-focused conferences and report their experiences in practitioner-oriented online platforms in the form of artifacts [22, 46]. With the help of an MLR, we are able to capture both: insights presented in academic conferences as well as insights presented in non-academic conferences. In this manner, the MLR complements the knowledge that a SLR provides. We still acknowledge the value of reviewing academic publications and that is why we review Internet artifacts as well as academic publications, which is a form of SLR. We also acknowledge that including Internet artifacts in the analysis can add bias to the derived results, which we mitigate using raters who read each Internet artifact to ensure the artifact of interest is in fact related to compiler defects.

We also report a comparison of the identified techniques as part of RQ3. We observe that if we considered only a review of academic publications we could have not known that commercial static analysis tool usage and user action are also used to identify defects in the compiler.

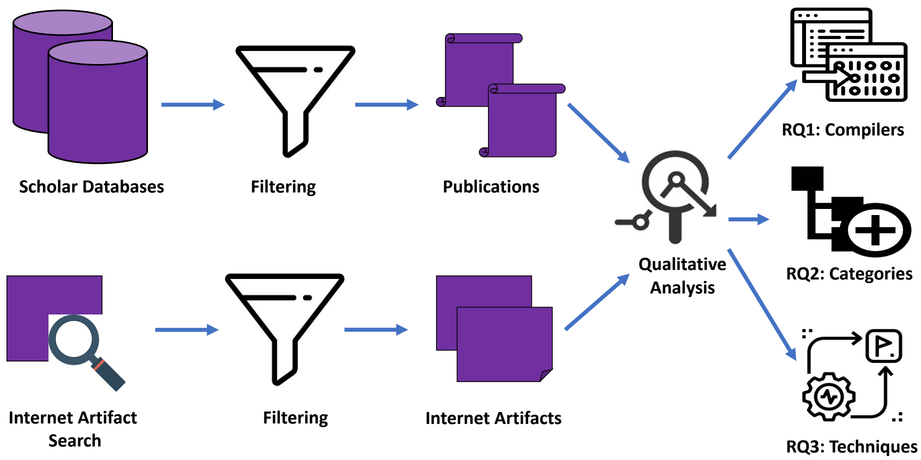

Fig. 1. An overview of our methodology.

We also would have not known that the techniques that are commonplace in academic peer-reviewed publications are not that commonly used by the practitioner community. For example, the techniques that are reported in peer-reviewed publications but not in Internet artifacts are address discrepancy analysis, deep learning, equivalence modulo input, Markov chains, optimization pattern synthesis, reinforcement learning, semantic specification, skeletal program enumeration, and tenor mutation. This indicates a gap between research and practice. Just by using SLR, we would have not learned this information.

In short, using MLR we can synthesize evidence from both types of artifacts: academic peer-reviewed publications and Internet artifacts that are not peer-reviewed. Therefore, MLR provides analysis that complements insights generated only by conducting an SLR.

Contributions: This work makes the following contributions:

- A list of defect categories for compilers derived from publications and Internet artifacts;
- A list of techniques used to identify defects in compilers as reported in publications and Internet artifacts; and
- A mapping between identified defect categories and the techniques used to identify defect categories.

We organize the rest of the paper as follows: we provide the methodology in Section 2. We report our findings in Section 3 and discuss these findings in Section 4. We discuss the limitations of our MLR and related work, respectively, in Sections 5 and 6. Finally, we conclude the paper in Section 7.

2 METHODOLOGY

We describe the methodology to conduct our MLR in this section. An MLR is a variant of systematic literature review that includes two types of resources: (i) Internet artifacts, such as blog posts and conference presentations, and (ii) peer-reviewed publications. Internet artifacts are an example of grey literature that has been well-regarded by literature review experts as an established source to obtain and synthesize practitioner perceptions [21]. According to Rainer et

Table 1. Criteria to Plan the MLR

Criteria	Third Author	Second Author
1. Is the subject complex and not solvable by considering only the formal literature?	**Yes**. Currently, available peer-reviewed publications have not synthesized existing literature related to compiler defects.	**Yes**. To date, no paper has systematically categorized defects in compilers.
2. Is there a lack of volume or quality of evidence or a lack of consensus on outcome measurement in the formal literature?	**Yes**. Internet artifacts, such as blog posts, tutorials, videos, and white papers, are prevalent compared to peer-reviewed publications in Kubernetes.	**Yes**. Peer-reviewed research lacks discussion of defect-related issues in compilers.
3. Is the contextual information important to the subject under study?	**Yes**. Understanding defects in compilers is important to build quality assurance into any software ecosystem.	**Yes**. Compiler defects are crucial to understanding how to integrate reliability into a software ecosystem.
4. Is it the goal to validate or corroborate scientific outcomes with practical experiences?	**Yes**. The goal is to compare the defect categories identified from Internet artifacts to that of peer-reviewed publications.	**Yes**.The goal of this research is to compare defects reported in peer-reviewed publications and Internet artifacts.
5. Is it the goal to challenge assumptions or falsify results from practice using peer-reviewed research or vice versa?	**No**. The goal is not to challenge current assumptions but to compare the defect categories studied by researchers and practitioners.	**No**. The goal of this research is not to challenge existing research related to compiler defects.
6. Would a synthesis of insights and evidence from the industrial and academic community be useful to one or even both communities?	**Yes**. A synthesis of insights and evidence from the industrial and academic community for compiler defects will help both communities.	**Yes**. Industry and academia would benefit from combining industry knowledge and academic knowledge related to compiler defects.
7. Is there a large volume of practitioner sources indicating high practitioner interest in a topic?	**No**. No such evidence was recorded.	**No**. We have not observed such evidence.

al. [47], with grey literature, such as with Internet artifacts practitioners provide stories, analogies, examples, and popular opinions as evidence, which they further use to justify their beliefs or refute existing beliefs. Use of internet artifact analysis has helped the software engineering community understand the best practices for contiguous deployment [46], devops security [59], securing Kubernetes installations [53], and managing secrets with secret management tools [41].

For the scope of our study, we define a compiler as a special type of software that takes source code as input and provides machine code or binary executables as input. This category of software can support multiple languages and have multiple compilation engines to support each of these languages. Furthermore, based on our definition, this type of software can provide interfaces to develop even more compilation units and provide rich software development experience so that along with generating machine code or binary executables, users can perform testing, linting, and version control.

In particular, we follow Garousi et al. [22]'s guidelines for conducting MLR. Figure 1 shows an overview of our methodology.

2.1 Plan for MLR

Garousi et al. [22] recommend that the researchers need to evaluate themselves if an MLR is appropriate for a specific research topic [22] before conducting the MLR. To that end, we use a set of criteria provided by Garousi et al. [22] that is listed in Table 1. If researchers involved in MLR agree on most of the criteria, then the researchers can move forward with the MLR. From Table 1 we observe the two researchers, i.e., the second and third authors of the paper, responded with 'Yes' for five of the seven criteria, and responded with 'No' for criteria #5 and #7.

2.2 Search for Internet Artifacts and Publications

We use two types of documents for our MLR: *first*, Internet artifacts, such as white papers, Slide share presentations [1], and blog posts. *Second*, we use peer-reviewed publications that have studied compiler defects. For collecting Internet artifacts, we use the Google search engine in incognito mode with a set of search strings. Following Kuhrmann et al. [29]'s guidelines we use five scholar databases, namely, (i) ACM Digital Library [2], (ii) IEEE Xplore [3], (iii) Springer Link [4], (iv) ScienceDirect [5], and (v) Wiley Online Library [6]. Kuhrmann et al. [29] recommend these scholar databases to use in systematic mapping studies and systematic literature reviews.

To identify Internet artifacts and peer-reviewed publications, we use a set of search strings that were derived using snowballing technique [60] following the guidelines of Garousi et al. [22]. To derive initial search strings, we first start with the search string 'compiler defect', which we use to collect the most relevant 100 Internet artifacts where relevance is determined by the Google search engine. Our assumption is that by using a set of 100 Internet artifacts, we will get the set of search keywords necessary to conduct our MLR. By reading each of these 100 Internet artifacts, the third author observes that while describing compiler defects, practitioners also use other terms. Considering these observations, we obtain a set of search strings that we use to identify Internet artifacts and peer-reviewed publications: `'buggy compiler'`, `'compiler' AND 'bug'`, `'compiler' AND 'defect'`, `'compiler' AND 'failure'`, `'compiler' AND 'fault'`, `'compiler' AND 'fuzzing'`, `'incorrect behavior' AND 'compiler'`, and `'miscompilation' AND 'bug'`.

For each search string, we collect the first 100 Internet artifacts provided by the Google search engine. From the five scholarly databases, we obtain 12,619 search results for the five search strings.

The focus of our paper is to find defect categories that have been reported in both: literature that is peer-reviewed and literature authored by practitioners that are not peer-reviewed. From our set of keywords, we are able to identify all 10 publications listed as part of a quasi-gold set.

2.3 Apply Inclusion and Exclusion Criteria

As both scholar databases and the Google search engine are susceptible to respectively yielding publications and Internet artifacts that are not relevant to an MLR, following Garousi et al.'s. [22] guidelines, we apply inclusion and exclusion criteria that are described below:

Exclusion Criteria: We exclude peer-reviewed publications and Internet artifacts that satisfy the following criteria:

- The artifact/publication is not written in English.
- The artifact/publication is not related to compiler error management. We exclude publications that discuss how developers comprehend and engage with compiler error messages as these publications do not discuss defects within the compiler.

[1] https://www.slideshare.net/
[2] https://dl.acm.org/
[3] https://ieeexplore.ieee.org/Xplore/home.jsp
[4] https://link.springer.com/
[5] https://www.sciencedirect.com/book/9781843341550/digital-libraries
[6] https://onlinelibrary.wiley.com/

For publication names returned by scholar databases, we apply an additional exclusion criterion: we exclude publications that are indexed in scholar databases but not peer-reviewed, such as keynote abstracts, call-for papers, and presentations.

Inclusion Criteria: We set the inclusion criteria for peer-reviewed publications and Internet artifacts as follows: (i) the artifact/publication is available for reading; (ii) the artifact/publication is not a duplicate. We determine an Internet artifact to be a duplicate of another if the title, content, and author(s) are the same as another Internet artifact. We randomly picked one of the duplicated Internet artifacts and included it in our set. We consider a pre-print as a duplicate. In the case of a journal publication that is an extension of a conference publication, we identify the conference and the journal paper as two separate publications; (iii) the artifact/publication is related to a compiler; and (iv) the content of the artifact/publications discusses defects that occur in a compiler. In the case of Internet artifacts, the second and third authors individually read the content of each Internet artifact to determine this criterion. In the case of peer-reviewed publications, the second and third authors individually read all the content of each paper to determine this criterion. For both cases, the authors determine if defects are discussed in the content. Both authors use the IEEE definition to determine the discussion of defects: "*An imperfection in a software artifact that needs to be repaired or replaced*".

The third author filters search results and identifies peer-reviewed publications written in English and available for reading. The third author retrieves 11,437 peer-reviewed publications from five scholarly databases. All the publications were available on December 2021. Upon applying our inclusion and exclusion criteria, the third author identifies 377 publications. At this stage, both the second and third authors read each of the 377 publications in detail and respectively identifies 27 and 37 publications to include a description of compiler defects.

The second and third authors disagreed on 28 publications. The Cohen's Kappa is 0.21, which is a 'fair' agreement [30]. The disagreements are resolved by the last author, and the last author's decision on the disagreed publications is final. Upon resolving all disagreements, we obtain a set of 26 peer-reviewed publications that we use in our MLR. Table **??** in Appendix (Section **??**) lists the publication titles. A complete breakdown of the publication search process is shown in Figure 2.

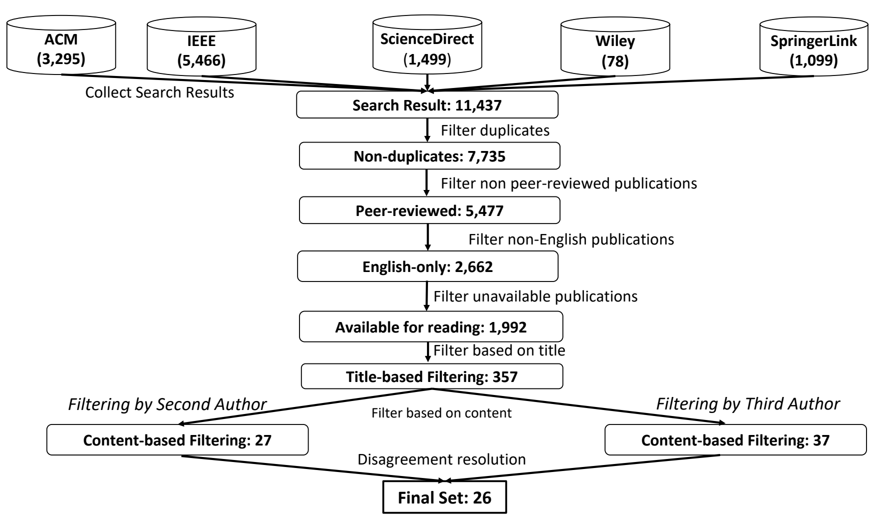

Fig. 2. Search and filtering of peer-reviewed publications to conduct our MLR.

Upon deriving the set of 26 publications, we validate our set of obtained publications by identifying if our set includes quasi-gold standard publications, i.e., publications that are well-regarded and deemed representative of compiler defect research. The third author, who has ten years of experience in software engineering research and is not involved in collecting this set of 26 publications, provided us with the quasi-gold set. The quasi-gold set includes the following publications: "Finding and Understanding Bugs in C Compilers" [62], "An Empirical Study of Optimization Bugs in GCC and LLVM" [67], " Well-typed Programs Can Go Wrong: A Study of Typing-related Bugs in JVM Compilers" [5], "Towards Understanding Tool-chain Bugs in the LLVM Compiler Infrastructure" [61], "Skeletal Program Enumeration for Rigorous Compiler Testing" [65], "Compiler Fuzzing Through Deep Learning" [12], and " Finding Compiler Bugs via Live Code Mutation" [56].

Our identified set of 26 publications includes all of these ten publications used in the quasi-gold set, which gives us the confidence that our collection of search strings is good enough to retrieve most of the relevant publications related to compiler defect characterization.

For Internet artifacts, both the second and third authors of the paper read each of the first 100 results from the Google Search for eight search strings. The search results are retrieved on January 2022. Initially, the third author removes duplicates, inspects availability, and removes non-English artifacts. This set of steps gives a total of 495 artifacts. Next, the third and second authors individually read each of the 495 artifacts and identified a set of 31 Internet artifacts and 28 Internet artifacts. The third and second authors disagreed on 23 Internet artifacts on their relationship with compiler defects. The Cohen's Kappa is 0.22, which is a 'fair' agreement [30]. The last author resolves the disagreements between the authors, whose decision is considered final. The last author is given a list of Internet artifacts for which the second and third authors disagreed. By reading the title and the content for each of the 23 Internet artifacts, the last author determines a set of 32 Internet artifacts that we use in our MLR. Table **??** in the Appendix (Section **??**) lists the 32 Internet artifact URLs. A complete breakdown of our search and filtering process to collect the Internet artifacts is shown in Figure 3. Ratings for all Internet artifact URLs and publication references used in the paper are publicly available online [40].

2.4 Assess Quality

Following guidelines from prior work [22, 27] we conduct a quality assessment of the collected Internet artifacts and peer-reviewed publications, respectively, in Sections 2.4.1 and 2.4.2.

2.4.1 Quality Assessment of Internet Artifacts. For the quality assessment of our set of 32 Internet artifacts, we use the assessment criteria provided by Garousi et al. [22]. Each of the assessment criteria is listed in Table 2:

We use a 3-point scale where '1.0' refers to 'yes'; 0.5 refers to 'partially'; 0.0 refers to 'no' for Q1-Q11. For Q12, we use a 3-point scale of 1.0, 0.5, and 0.0 to refer to high, moderate, and low credibility. The second and third authors individually read all of the 314 Internet artifacts to determine a value for Q1-Q12. Then, we report the average of the scores reported by the second and third authors.

A summary of the average rating for each of the questions for the 32 Internet artifacts is given in Table 3. The detailed rating for each of the Internet artifacts is available in Table **??** of the Appendix (Section **??**), where each cell represents the ratings obtained by the second and third authors. From Table 3 we observe Internet artifacts to score >= 0.5 for reputation (Q1), aim (Q3), coverage (Q5), objectivity (Q6), links to important literature (Q9), impact (Q11), and credibility

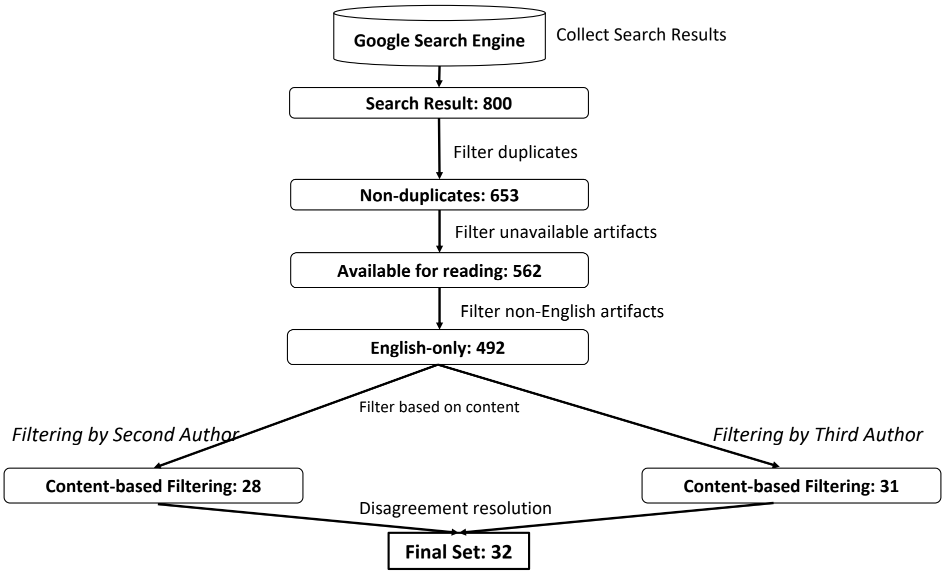

Fig. 3. Search and filtering of Internet artifacts to conduct our MLR.

Table 2. Quality Assessment Criteria for Internet Artifacts

Criterion	Question
Criterion-1: Reputation	**Q1:** Is the publishing organization reputable? **Q2:** Is an individual author associated with a reputable organization?
Criterion-2: Methodology (Aim, Reference, Coverage)	**Q3:** Does the source have a clearly stated aim? **Q4:** Is the source supported by authoritative, contemporary references? **Q5:** Does the work cover a specific question?
Criterion-3: Objectivity	**Q6:** Is the statement in the sources as objective as possible? Or, is the statement a subjective opinion? **Q7:** Is there a vested interest? For example, a tool comparison by authors working for a particular tool vendor.
Criterion-4: Date	**Q8:** Does the item have a clearly stated date?
Criterion-5: Position with respect to related sources	**Q9:** Have key related Internet artifacts or peer-reviewed publications been linked to or discussed?
Criterion-6: Novelty	**Q10:** Does it strengthen or refute a current position? Does it advance a new position?
Criterion-7: Impact	**Q11:** What is the impact of the Internet artifact? The raters apply subjective evaluation to determine the impact of an Internet artifact. The rater considers the following concepts to determine impact: count of backlinks, count of comments, count of views, and count of shares.
Criterion-8: Credibility	**Q12:** What is the credibility of the Internet artifact? (i): High credibility: Books, magazines, thesis documents, government reports, white papers; (ii) Moderate credibility: Annual reports, news articles, presentations, videos, Q/A sites (e.g. StackOverflow), Wikipedia articles; (iii) Low credibility: Blogs, emails, tweets.

(Q12). which corresponds to the date of the Internet artifact. The range of scores for each criterion is presented as minimum and maximum in the 'Min, Max' column.

2.4.2 Quality Assessment of Publications. We follow the criteria provided by Kitchenham et al. [27] to assess the quality of a peer-reviewed publication. A higher-quality score indicates that the publication clearly describes the goal, contains actionable results, clearly discusses the limitations, and contains a clear presentation structure. The criteria set that we use for our set of peer-reviewed publications is listed in Table 4.

Table 3. Quality Assessment of Internet Artifacts Related to Compiler Defects

Criterion	Average Rating	Min, Max
Q1 (Reputation of Publishing Organization)	0.5	0.0, 1.0
Q2 (Reputation of Author's Organization)	0.3	0.0, 1.0
Q3 (Clearly Stated Aim)	0.6	0.0, 1.0
Q4 (References)	0.4	0.0, 1.0
Q5 (Coverage)	0.6	0.0, 1.0
Q6 (Content Objectivity)	0.5	0.0, 1.0
Q7 (Vested Interest)	0.4	0.0, 1.0
Q8 (Clearly Stated Date)	0.1	0.0, 1.0
Q9 (Links to Important Literature)	0.8	0.0, 1.0
Q10 (Strengthen/Refute Position)	0.3	0.0, 1.0
Q11 (Impact)	0.5	0.0, 1.0
Q12 (Credibility)	0.5	0.0, 1.0

Table 4. Quality Assessment Criteria for Peer-reviewed Publications

Criterion	Description
Q1 (Aim)	Do the authors clearly state the aim of the research?
Q2 (Units)	Do the authors describe the sample and experimental units?
Q3 (Design)	Do the authors describe the design of the experiment?
Q4 (Data Collection)	Do the authors describe the data collection procedures and define the measures?
Q5 (Data Analysis)	Do the authors define the data analysis procedures?
Q6 (Bias)	Do the authors discuss potential experimenter bias?
Q7 (Limitations)	Do the authors discuss the limitations of their study?
Q8 (Clarity)	Do the authors state the findings clearly?
Q9 (Usefulness)	Is there evidence that the Experiment/Quasi-Experiment can be used by other researchers/practitioners?

Table 5. Quality Assessment for 26 Publications

Criterion	Average Rating	Min, Max
Q1 (Aim)	3.7	3.5, 4.0
Q2 (Units)	3.0	1.5, 4.0
Q3 (Design)	3.7	2.5, 4.0
Q4 (Data Collection)	2.6	1.5, 4.0
Q5 (Data Analysis)	2.3	1.0, 4.0
Q6 (Bias)	1.6	1.0, 2.5
Q7 (Limitations)	1.9	1.0, 4.0
Q8 (Clarity)	3.3	2.5, 4.0
Q9 (Usefulness)	2.2	1.5, 4.0

We follow the procedure used by Kitchenham et al. [28] to resolve disagreements. For the resolution of disagreements, we compute the average of the scores reported by both raters.

After answering each of the above nine questions, we provide a rating score associated with each of the answers between 1 and 4. The rating 1 implies 'not at all'; 2 implies 'somewhat'; 3 implies 'mostly,'; and 4 implies 'fully'. As the rating process of the research articles is subjective, we assign two raters, i.e., the second and third authors, who independently provide a rating to each publication. We report the average rating score of both raters for each publication. We summarize the average rating of the quality assessments for the 26 publications in Table 5 and the quality assessment rating for each of the peer-reviewed publications is described in Table **??** of the Appendix (Section **??**). From Table 5, we observe publications related to compiler defects to score > 3.5 for aim and clarity-related discussion on average. With respect to the discussion of bias and limitations, the set of 26 publications scores < 2.0.

2.5 Answer to Research Questions

We provide the methodology to answer our research questions in this section.

2.5.1 Methodology to Answer RQ1. The first and third author individually reads each of the 32 Internet artifacts and 26 publications to identify the compilers that have been addressed. For each artifact and publication, the third author documents the date of the artifact and publication, the specific compiler that has been addressed, and the programming language that corresponds to the compiler.

2.5.2 Methodology to Answer RQ2. We answer RQ2 by applying a qualitative analysis technique called open coding [48]. Open coding helps researchers to summarize the underlying theme from unstructured text [48]. We hypothesize that by applying open coding, we can group defects that have been reported in our set of artifacts and publications. The third author performs open coding with the content from artifacts and peer-reviewed publications. Upon completion, the third author derives a list of defect categories for compilers as reported in artifacts and publications.

Rater Verification: The open coding process is susceptible to rater bias, which we mitigate by using the second author to perform rater verification. The second author was not involved in the open coding process. As part of rater verification, the second author performs closed coding [48], using which the author maps an identified defect category to each of the 26 peer-reviewed publications and 32 Internet artifacts. We do not impose any time limit on the rater to perform verification.

Upon completion, we record a Cohen's Kappa of 0.83 and 0.86, respectively, for Internet artifacts and peer-reviewed publications between the second and third authors. For both artifacts and publications, the agreement is 'substantial' [30] between the second and third authors.

Mapping of Defect Categories and Compiler Components: We further investigated which of the identified defect categories are applicable to a component of a compiler. The purpose of this investigation is to generate insights into what components are likely to include certain defect categories. For our investigation, we leverage the typical compiler components described by Aho et al. [2], and summarized in Figure 4. Each compiler takes a computer program as input and generates code that is executable on a target machine.

Aho et al. [2] lists the following components for a compiler that is shown in black ink in Figure 4:

- Lexical analyzer: This component of the compiler parses the program into a sequence of tokens.
- Syntax analyzer: This component of the compiler takes the output of the lexical analyzer and applies grammar to determine if the computer program satisfies the syntactical rules of the programming language.
- Semantic analyzer: This component of the compiler uses the output of the syntax analyzer in the form of abstract syntax trees as input, and checks whether the computer program is semantically consistent with language definition.
- Intermediate representation generator: This component of the compiler uses the output of the semantic analyzer as input to generate an intermediate representation that is in between source code and machine code in terms of representations.
- Code optimizer: This component of the compiler uses the intermediate code to perform optimizations so that the computer program upon execution consumes lesser resources, such as CPU and memory.

- Code generator: This component of the compiler converts the optimized intermediate code into machine code so that the computer program can be executed by the computing system, e.g., an x86 processor.

As part of this investigation, we read the defect categories and corresponding examples to determine if a defect category occurs for one or multiple components of the compiler. We repeat the procedure for both Internet artifacts and publications.

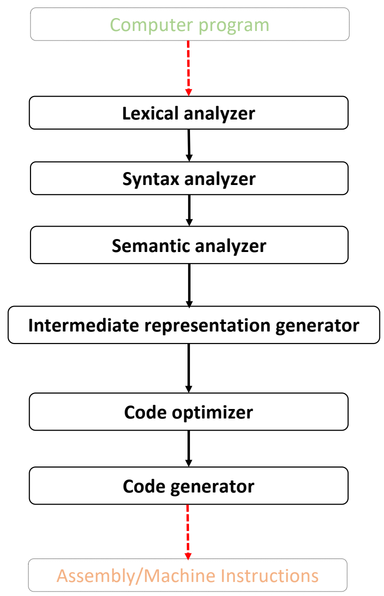

Fig. 4. Components of a typical compiler as summarized by Aho et al. [2].

2.5.3 Methodology to Answer RQ3. RQ3 focuses on the techniques that have been used to identify defects in a certain compiler. Answers to this research question can aid practitioners and researchers in understanding the techniques that are used to find defects in a compiler and apply that understanding to identify defects in compilers that remain under-explored to date. To answer RQ3, the third author reads each artifact and publication, respectively, in our sets of 32 Internet artifacts and 26 publications. The third author separates publications that clearly describe a technique that is used to identify defects in a compiler. The third author applies the same procedure for Internet artifacts.

3 RESULTS

We provide answers to our research questions in this section. We answer our research questions by analyzing 32 Internet artifacts and 26 peer-reviewed publications. Temporal trends of Internet artifacts and publications are respectively, shown in Figure 5 and 6.

3.1 Answer to RQ1

In this section, we answer *RQ1: Which compilers have been studied in Internet artifacts and peer-reviewed publications that have investigated defects in compilers?* We provide the count of Internet artifacts and peer-reviewed publications in

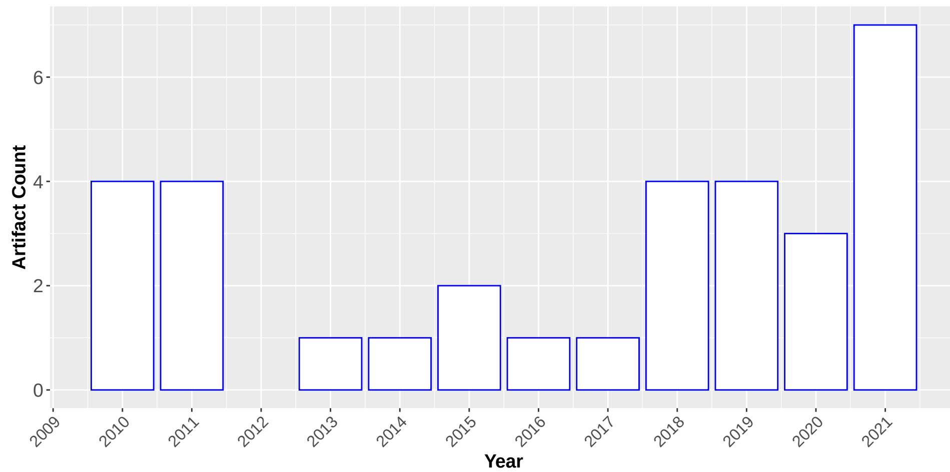

Fig. 5. Temporal trends of the 32 Internet artifacts.

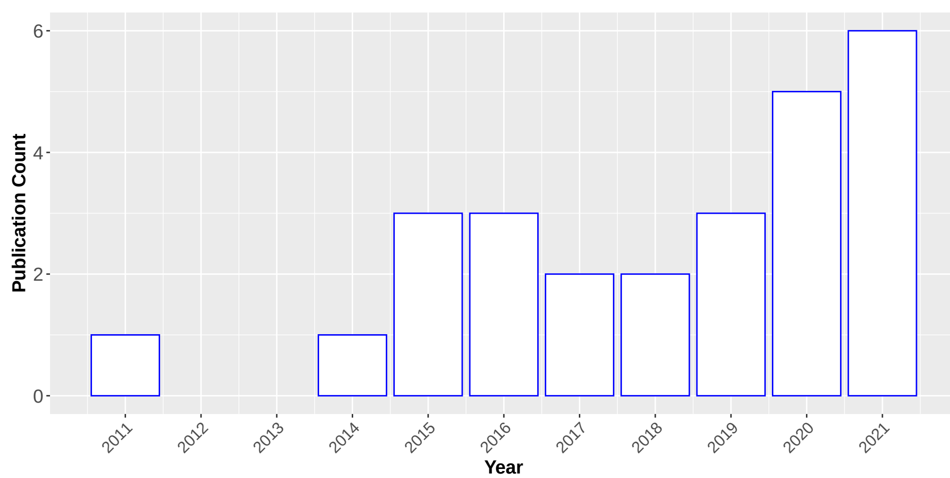

Fig. 6. Temporal trends of the 26 peer-reviewed publications.

which a compiler has been discussed in the context of compiler defects. In Tables 6 and 7 we respectively, provide the count of publications in which a compiler has been addressed.

We observe similarities and differences with respect to studied compilers as documented in Table 6 and 7. GCC is the most frequently studied compiler in our set of Internet artifacts, followed by LLVM. In the case of publications, GCC and LLVM are the most frequently mentioned compilers. Certain compilers are only studied in artifacts: GNU Fortran, Intel Fortran, .NET Fortran, PGI Fortran, Cray Compiling Environment [7], Xilinx SDK, and CraneLift, the WebAssembly Compiler. Compilers that are only studied in publications and not in Internet artifacts are Simulink, V8 Javascript,

[7]https://docs.lumi-supercomputer.eu/development/compiling/cce/

Table 6. Compilers Discussed in Our Set of 32 Artifacts

Compiler	Artifact Index	Count
GCC	IA4, IA5, IA6, IA8, IA9 , IA14 , IA16, IA18, IA19, IA21, IA22, IA23, IA24, IA25, IA26, IA28	16
LLVM	IA6, IA4, IA17, IA20	4
Arduino SDK	IA13, IA14	2
GNU Fortran Compiler	IA2, IA27	2
Intel Fortran Compiler	IA2, IA27	2
Java 7 Compiler	IA1, IA26	2
PGI Fortran Compiler	IA2, IA27	2
.NET	IA10, IA32	2
Clang	IA6	1
Code Composer Studio	IA11	1
Cray Compiling Environment	IA3	1
IBM Fortran Compiler	IA27	1
Intel C++ Compiler	IA6	1
Kotlin	IA15	1
Solidity	IA31	1
WebAssembly Compiler (CraneLift)	IA30	1
XCode	IA20	1
Xilinx SDK	IA12	1

Table 7. Compilers Discussed in Our Set of 26 Publications

Compiler	Publication Index	Count
GCC	P1, P5, P9 , P10, P12, P15, P16, P18, P20, P22, P23	11
LLVM	P1, P4, P10, P12, P15, P16, P17, P18, P20, P22, P23	11
Clang	P4, P9, P16, P18	4
Simulink	P8, P14, P19	3
V8 Javascript	P3, P22, P26	3
ChakraCore	P22, P26	2
Javascript Core	P22, P26	2
OpenCL	P7, P11	2
Kotlin	P6, P21	2
RustC	P18, P24	2
Bambu	P13	1
Commercial HLS Compiler	P13	1
Glow	P2	1
GraalJS	P26	1
Groovy	P6	1
Hermes	P26	1
Intel C++	P23	1
JerryScript	P26	1
K-Java	P19	1
KSolidity	P19	1
Legup	P13	1
Nashorn	P26	1
nGraph	P2	1
OpenJDK	P6	1
P4	P25	1
QuickJS	P26	1
Rhino	P26	1
Scala	P6	1
SpiderMonkey	P26	1
Turbofan	P3	1
TVM	P2	1

ChakraCore, JavascriptCore, OpenCL, RustC, Bambu, Commercial HLS Compiler, Glow, GraalJS, Grrovy, Hermes, Jerryscript, K-Java, KSolidity, Legup, Nashorn, nGraph, OpenJDK, P4, QuickJS, Rhino, Scala, SpiderMonkey, Torubofan, and TVM. Our findings show a disconnect between the compilers that are studied in peer-reviewed publications and what practitioners are discussing and reporting.

Answer to RQ1: A wide range of compilers have been investigated in prior work, such as GCC, LLVM, and deep learning compilers. GCC is the most frequently mentioned compiler amongst artifacts as well as in peer-reviewed publications.

3.2 Answer to RQ2

We provide answers to *RQ2: What categories of defects have been reported in Internet artifacts and peer-reviewed publications that have investigated defects in compilers?* in this section.

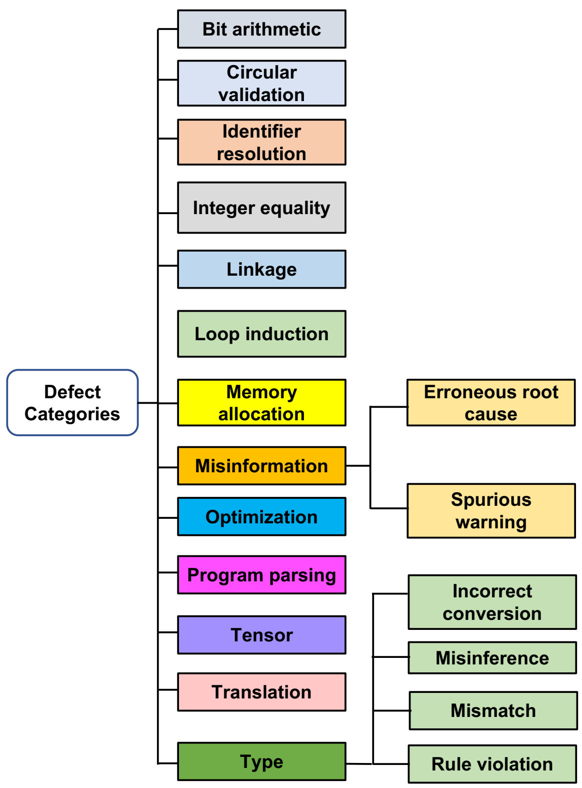

Fig. 7. Defect categories identified from our MLR.

We identify 13 categories of defects that are shown in Figure 7, which we describe below:

Bit Arithmetic Defects: This category of defects occurs when a compiler does not adequately implement bit arithmetic. This category of defects has been reported both in Internet artifacts as well as peer-reviewed publications.

Example: In an artifact [25], a bit arithmetic defect was reported for Cranelift, a WebAssembly compiler. The defect occurred because of interpreting a '4GB' parameter as 4,000,000,000 bytes in decimal gigabytes. The maximum heap size was configured below 4GiB, "4,294,967,296", which made some unexpected instructions while investigating the disassembly code shown in Listing 1. The WebAssembly compiler's `load` and `store` instructions include an offset immediate, which was designed to simplify loads and stores in working with structures. However, this allows any user to avoid the bounds check by using a heap offset that is low, then adding a large offset in a load or store, eventually allowing the program to enter a region just before an instance's heap, which could have serious consequences. The code snippet in Listing 1 shows how a bit of arithmetic defect can occur. The defect resulted in a crash.

```
mov    edi, 0xee6b27fe          ; an entirely unexpected constant: 3,999,999,998
movsxd rax, DWORD PTR [rsp+0x88] ; the incorrect sign-extended load
cmp    eax, edi                 ; compare against the heap bound
jae    ff0 <guest_func_4+0x360>  ; and branch to a trap site if out of bounds
```

Listing 1. Example of a bit arithmetic defect reported for the WebAssembly compiler.

Circular Validation Defects: This category of defects occurs when there are no checks for the presence of circular dependencies between objects or variables.

Example: As shown in Listing 2, Chaliasos et al. [5] reported an absent circular validation defect for Scalac, the compiler for Scala. The two classes A and B are defined with a circular dependency issue. When Scalac checks the correctness of these declarations, it does not discover this dependence problem. As a result, it crashes when Scalac unboxes these value classes depending on the types.

```
case class A(x :B) extends AnyVal;
case class B(x :A) extends AnyVal;
```

Listing 2. Example of an absent circular validation defect that occurs for Scalac.

Identifier Resolution Defects: This category of defects occurs when a compiler fails to resolve an identifier name to its corresponding definition or scope.

Example: As shown in Listing 3, Chaliasos et al. [5] reported an identifier resolution defect for Java, the Java compiler. The method error defined in line 7 is the most particular because its signature is less generic than the signature of the error specified in line 6. Because an identifier resolution in Javac fails to resolve the identifiers in lines #6 and #7 adequately, it identifies both methods as ambiguous. The program does not get compiled even though it is a syntactically valid program.

```
1 class Test {
2   void test() {
3     Exception ex = null;
4     error("error", ex);
5   }
6   void error(Object o, Object... p) {}
7   void error(Object o, Throwable t, Object... p) \{}
8 }
```

Listing 3. Example of an identifier resolution defect that occurs for Javac.

Integer Equality Defects: This category of defects occurs when a compiler does not adequately check for integer equality.

Example: As shown in Listing 4, Zhang et al. [65] reported an integer equality defect for GCC. The defect occurred for not checking for Integer equality via value comparison, which violated an assertion. The defect resulted in a GCC crash and was repaired by using value comparison to check integer equality.

```
struct s { char c[1]; };
struct s a, b, c;
int d; int e;
void bar (void)
{
    e ? (d==0 ? b : c).c : (d==0 ? b : c).c;
}
```

Listing 4. Example of an integer equality defect that occurs for GCC.

Linkage Defects: This category of defects occurs due to unsuccessful linkages between components of a compiler. The linkage defect category is not limited to the linker, i.e., the software that takes one or more object files and combines them into a single executable file, library file, or another object file [14]. This category of defects can occur when linkages are established between one component to another within a compiler.

Example: A linkage defect was reported for LLVM on Xcode while using the LLVM component called 'lldb' [61]. 'lldb' is a native debugger that is available as part of the LLVM compiler toolchain. It is more memory efficient and faster than gdb, the GNU project debugger [61]. The defect occurs when the object method needs to be called in an undefined entity [8]. Listing 5 shows the error message for the defect. The defect is repaired by adding a link to the object method in a configuration file essential to the lldb component of LLVM.

```
build/Release+Asserts/x86_64/lib/libclang_rt.cc_kext_i386_osx.a, file was built for archive which is not the architecture being linked (x86_64): /Users/buildslave/jenkins/lldb/llvm-build/Release+Asserts/x86_64/lib/libclang_rt.cc_kext_i386_osx.a
Undefined symbols for architecture x86_64:
  "PDBASTParser::~PDBASTParser()", referenced from:
      lldb_private::ClangASTContext::ClangASTContext(char const*) in liblldb-core.a(ClangASTContext.o)
      lldb_private::ClangASTContext::~ClangASTContext() in liblldb-core.a(ClangASTContext.o)
ld: symbol(s) not found for architecture x86_64
```

Listing 5. Error message for a linkage defect in LLVM.

Loop Induction Defects: This category of defects occurs when a compiler's loop induction procedure is incorrect. As part of the loop induction procedure, a compiler checks for loop invariants in the case of computer programs that use recursions or iterations. Loop invariants are used to determine the progress or completion time of a computer program [36, 38]. Loop induction defects are different from optimization defects as loop induction defects are related to program invariants that can occur with or without the use of optimization flags.

Example: As shown in Listing 6, Yang et al. [62] reported a loop induction defect for LLVM. When the `-indvars` flag is used for LLVM the code in line#5 (`if (x) break ;`) makes LLVM conclude that x is 1 after loop is executed, instead of printing 5.

[8] https://bugs.llvm.org/show_bug.cgi?id=27362

```
Void ; foo(void){
 int x;
 for (x = 0; x < 5; x++) \{
    if (x) ; continue;
    if (x) ; break;
 }
 printf("%d", x);
}
```

Listing 6. Example of a loop induction defect that occurs for LLVM.

Invalid Memory Access Defects: This category of defects occurs when a program attempts to access a memory location that is not allowed to access or tries to access a memory location in such a way that is not allowed. This category of defects has been reported in Internet artifacts.

Example: In an artifact [63], an invalid memory access defect occurs for the Fortran compiler when the following code snippet is executed. The defect resulted in a segmentation fault.

```
 PBL_THICK(-1000000,J)    = BLTHIK
```

Listing 7. Example of an invalid memory access defect reported for the Fortran compiler.

Misinformation Defects: This category of defects occurs when the compiler fails to provide adequate information to the developer on how to fix a compiler error or a warning. We identify two sub-categories:

Erroneous root cause: Defects that do not adequately identify the root cause of a compilation error or a compiler warning.

Example: In an artifact [50], a misinformation defect occurred when using the PGI 14.1 Fortran compiler. The defect occurs from not providing the correct information that caused the defect. The defect occurred for the program presented in Listing 8. The module `test_types` is invalid because of the subroutine `do_nothing()` not accepting a `class(foo)` argument. Instead of providing this information, the compiler generates a segmentation fault leaving no clues for a developer on how to fix the issue.

```
module test_types

type :: foo
 contains
   procedure :: do_nothing
end type foo

contains

  subroutine do_nothing()
  end subroutine do_nothing

end module test_types
```

Listing 8. Example of an erroneous root cause defect reported for the Fortran compiler.

Spurious Warning: Defects that occur because of the compiler's erroneous warning mechanisms that prevent a developer from identifying the location of a compiler warning.

Example: As shown in Listing 9, a spurious warning defect was reported for GCC [56]. GCC is expected to give a warning because the format string `s` is not null-terminated, and the `printf` function outputs the truncated string. Because of a defect, the warning is not reported.

```
void fn() { const char s[1] = "format"; printf(s); }
```

Listing 9. Example of a spurious warning defect reported for GCC.

Optimization Defects: This category of defects occurs when any undesired behavior occurs because of compiler optimization. Compiler optimization is a procedure where algorithms take a program to transfer it in such a way that it will execute the same output program but will use fewer resources or execution will be faster. This category of defects has been reported in Internet artifacts and peer-reviewed publications.

Example: In a Stack Overflow post [1], we document an example of an optimization defect. The defect occurs when the GCC compiler performs optimization that results in an infinite loop.

```
for (i = 1; i > 0; i += i) ++j;
```

Listing 10. Example of an optimization defect reported for the GCC compiler.

Program Parsing Defects: This category of defects occurs when the compiler fails to parse a computer program adequately. This category of defects has been reported in Internet artifacts and peer-reviewed publications.

Example: In an artifact [19], a program parsing defect occurred when using the Cray Compiling Environment. The defect occurred by using a Fortran-reserved keyword as a variable name. The Cray Compiling Environment incorrectly parsed `integerfoo` as a reserved keyword instead of a variable name. The defect resulted in a compiler error.

```
program main
implicit none

type integerfoo
  real :: bar
end type integerfoo

type(integerfoo) :: test

end program main
```

Listing 11. Example of a program parsing defect reported for the Fortran compiler.

Tensor Defects: This category of defects occurs when a compiler incorrectly computes tensors, which are used to implement deep learning algorithms.

Example: As shown in Listing 12, Shen et al. [54] reported a Tensor defect occurred because of Tensor shapes being incorrectly calculated by TFLite, a lightweight deep learning compiler available as part of the Tensorflow project. The defect was repaired by providing the correct batch size with `target_shape = tuple((-1, weight_tensor_shape[1]))`.

```
- input_size =1
- for _, shape in enumerate(input_tensor_shape):
-     input_size*=shape
- batch_size = int(intput\_size / weight_tensor_shape[1])
- target_shape = tuple((batch_size, weight_tensor_shape[1]))
+ target_shape = tuple((-1, weight_tensor_shape[1]))
```

Listing 12. Example of a Tensor defect that occurs for TFLite.

Translation Defects: This category of defects occurs when a compiler does not adequately translate the source code of a computer program into intermediate forms or binaries. This category of defects has been reported in Internet artifacts and peer-reviewed publications.

Example: In an artifact [13], a translation defect was reported for LLVM as shown in Listing 13. The defect occurs because of translating two static functions with the same names, both of whom define a lambda function. During the translation process, because of using lambda with `async`, the generated binary will have one symbol and will result in a crash.

```
template $<typename T>$ auto async() \{
      return [](auto func) \{
        [func] { func(); }();
      };
    }
    static void f(){
      async $<int>$()([] \{});
    }
    void f1() { f(); }
```

Listing 13. Example of a translation defect that occurs for LLVM.

Type Defects: This category of defects occurs when a compiler inadequately handles the program types. This category of defects has been reported in peer-reviewed publications. The category includes four sub-categories:

Incorrect conversion: Compiler defects that occur when the compiler incorrectly converts types.

Example: As shown in Listing 14, Chaliasos et al. [5] reported an example where types `A` and `B` needed to be converted to type `C`, but Kotlinc failed to do such.

```
interface A
interface B
class c: A, B
fun <T> T.m(): Unit where T: A, T:B {}
fun main(){
      c().foo()
}
```

Listing 14. Example of a an incorrect type conversion defect that occurs for Kotlinc.

Misinference: Compiler defects that occur when incorrect types are inferred for a variable or a function.

Example: Chaliasos et al. [5] reported a type misinference defect for Kotlinc, the Kotlin compiler. The defect occurs due to incorrect handling of function references, which eventually caused Kotlinc to construct a constraint problem with incomplete constraints. In Listing 15, the inference engine stops Kotlinc from instantiating the type variable `T` declared in class `A`.

```
class A<T>(val f:T)
fun test()\{
listOf<string>().map(::A)
}
```

Listing 15. Example of a type misinference defect that occurs for Kotlinc.

Mismatch: Compiler defects that occur when one program within the compiler fails to provide the correct type to another program.

Example: Shen et al. [54] reported a defect related to type mismatch for PyTorch. The output tensor type for the operator is expected to be Float32. However, the analogous Glow operator produces Float16. The defect was repaired by using an `upcast` operator as shown in Listing 16.

```
- return addValueMapping(output[0], EB->getResult());
+ if(is4Bit){
+   auto *CT = F.createConvertTo(
+   "ConvertEmbeddingBag4BitRowwiseOffsetsOutput"
+       EB.Elemkind::FloatTy);
+   retun addValueMapping(output[0], CT->getResult());
+ } else{
+   return addValueMapping(output[0], EB->getResult());
+ }
```

Listing 16. Example of a type mismatch defect that occurs for PyTorch.

Rule violation: Compiler defects that occur when the compiler violates the rules for the language's type system.

Example: As shown in Listing 17, Chaliasos et al. [5] reported an example where violation of type rule occurs. For the code snippet, Javac does not adhere to Java's type rules that result in considering c<? > to be a subtype of $I <?\ \mathit{extends}\ X, X >$.

```
Interface; I <X1, X2> {}
class ; C<T> implements ; I<T, T> {}
public ; class ; test{
   <X> void ; m(I<? ; extends ; X, X> arg) {}
   void ; test(c<?> arg){
       m(arg);
    }
}
```

Listing 17. Example of a type rule violation defect that occurs for Javac.

Mapping of Defect Categories with Artifacts and Publications: We provide a mapping between each identified defect category and the corresponding artifact in Table 8. The defect categories that we did not find in any of our

Table 8. Mapping Between Internet Artifacts and Defect Categories

Category	Artifact Index	Count
Bit arithmetic	IA30	1
Invalid Memory Access	IA2	1
Misinformation	IA2, IA9, IA10, IA12,IA13, IA16, IA20, IA23, IA25, IA26, IA27	11
Optimization	IA1, IA2, IA5, IA6, IA7, IA8, IA11, IA14, IA15, IA21, IA22, IA24, IA27, IA29, IA32	15
Program parsing	IA3, IA19	2
Translation	IA17, IA18, IA28	3
Type	IA31	1

Internet artifact sets are circular validation, identifier resolution, integer equality, linkage, loop induction, and tensor. The defect category that we observe in Internet artifacts but not in publications is invalid memory access. We also provide a mapping between each defect category and the corresponding publications in Table 9.

Table 9. Mapping Between Publications and Defect Categories

Category	Publication Index	Count
Bit arithmetic	P1, P13, P20	3
Circular Validation	P6	1
Identifier Resolution	P6	1
Integer equality	P16	1
Linkage	P17	1
Loop Induction	P5, P17	2
Invalid Memory Access	P12, P22	2
Misinformation	P6, P8, P9, P10, P14, P21	6
Optimization	P1, P3, P5, P10, P11, P15, P18, P22	8
Program parsing	P7, P18	2
Tensor	P2	1
Translation	P4, P6, P11,P12, P16, P19, P22, P23	8
Type	P2, P6, P17, P21, P22, P24, P25, P26	8

Mapping of Defect Categories with Compiler Characteristics: We also study the characteristics of the compilers for which we documented the identified bug categories. We summarize our results in Tables 10 and 11. The tables are sorted alphabetically based on the compiler name.

Each row lists a compiler and the defect categories that are associated with the compiler as shown in the 'Category' column. We further report the associated language, generated output type, and whether or not the compiler is open or closed source. For example, for the 'Bambu' compiler we record the bit arithmetic defect. The compiler is used for high-level synthesis (HLS) language, which generates hardware specification as output. The compiler is a closed source.

From Table 10 we observe defect categories to be diverse for open-source compilers compared to that of closed-source compilers. Certain defect categories are common across multiple types of compilers. From Table 11 we observe multiple Fortran-related compilers being studied for which practitioners have reported multiple defect categories.

Benchmarks reported in peer-reviewed publications: We report the benchmarks that have been used in our studied publications in Table 12. The 'Benchmark' column in Table 12 reports the benchmarks that have been used by each publication. If a publication does not report any benchmarks, then we report 'No benchmark reported'. We observe GCC and LLVM to be the most frequently used benchmarks in academic publications related to compiler defects.

Mapping of Defect Categories to Compilation Steps and Defect Categories: we provide a mapping between compilation steps and identified defect categories in Table 13.

Table 10. Mapping Between Defect Categories and Compiler Characteristics Based on Publications

Category	Compiler Name	Languages Used	Output	Open/Closed
Bit arithmetic	Bambu	HLS	Hardware Specifications	Closed
Type	ChakraCore	Javascript	Machine Code	Open
Integer equality, Misinformation, Optimization, Translation	Clang	C/C++, Objective C/C++, RenderScript	Machine	Open
Bit arithmetic	Commercial HLS Compiler	HLS	Hardware Specifications	Closed
Integer equality, Misinformation, Optimization, Translation	Clang	C/C++, Objective C/C++, RenderScript	Machine	Open
Circular validation, Identifier resolution, Misinformation, Translation, Type	Dotty	Scala 3	Java Bytecode	Open
Integer equality, Loop induction, Bit arithmetic, Invalid Memory Access, Misinformation, Optimization, Translation	GCC	C/C++	Binary/Assembly	Open
Tensor, Type	Glow	Dataflow graph	Machine	Open
Circular validation, Identifier resolution, Misinformation, Translation, Type	Groovy	Groovy	Java Bytecode	Open
Circular validation, Identifier resolution, Misinformation, Translation, Type	Kotlin	Kotlin	Java Bytecode	Open
Misinformation	K-Java	Java	Java	Open
Misinformation	KSolidity	Solidity	Java	Open
Bit arithmetic	Legup	HLS	Hardware Specifications	Open
Bit arithmetic, Linkage, Loop induction, Invalid Memory Access, Optimization, Translation, Type	LLVM	C/C++, C#, OpenCL, Ruby, Scala	Assembly	Open
Tensor, Type	nGraph	ONNX graph	Machine	Open
Program parsing	OpenCL	C/C++	Assembly	Open
Circular validation, Identifier resolution, Misinformation, Translation, Type	Open JDK	Java	Java Bytecode	Open
Circular validation, Identifier resolution, Misinformation, Translation, Type	Scala	Scala 2	Java Bytecode	Open
Optimization, Program parsing	RustC	Rust	Assembly	Open
Misinformation, Translation	Simulink	Simulink	Specification	Closed
Tensor, Type	TVM	Python deep learning	IR	Open
Optimization	TurboFan	Javascript	Machine	Open
Optimization, Type	V8 Javascript	Javascript	Machine	Open
Optimization, Type	Intel C++	C++	Machine	Open
Type	Javascript Core	Javascript	Machine Code	Closed
Program parsing, Type	RustC	Rust	Binary	Open
Type	P4 Compiler	P4 Programs	C/JSON/Machine Code	Open
Type	ChakraCore	Javascript	Machine Code	Open
Type	SpiderMonkey	Javascript	Byte Code	Open
Type	Rhino	Javascript	Nyte Code	Closed
Type	Nashorn	Javascript	Byte Code	Closed
Type	Hermes	Javascript	Byte Code	Open
Type	JerryScript	Javascript	Byte Code	Open
Type	QuickJS	Javascript	Binary	Open
Type	GraalJS	Javascript	Byte Code Code	Open

We further investigate if the identified defect categories appear for other software systems. As part of this review activity, we reviewed prior work on defect categorization and identified if one or multiple defect categories identified from our MLR also appear for other software systems. By reviewing these papers, we assume to identify defect categories that are applicable to other software systems. The papers that we reviewed are: “An Empirical Study on TensorFlow Program Bugs” [66], “Bug Characteristics in Open Source Software” [57]”, “Orthogonal Defect Classification: A Concept for In-process Measurements” [8]”, “Not All Bugs Are The Same: Understanding, Characterizing, and Classifying Bug Types” [4]”, “Defect Categorization: Making Use of a Decade of Widely Varying Historical Data” [52]”, “Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts” [42], “IoT Bugs and Development Challenges” [34]”, “Taxonomy of Real Faults in Deep Learning Systems” [24].

Table 11. Mapping Between Defect Categories and Compiler Characteristics Based on Artifacts

Category	Compiler Name	Languages Used	Output	Open/Closed
Optimization	.NET	C#, F#, Visual Basic	Machine	Open
Misinformation, Optimization	Arduino SDK	C++	Machine	Open
Optimization	Clang	C/C++, Objective C/C++, RenderScript	Machine	Open
Optimization	Code Composer Studio	C/C++	Assembly	Closed
Misinformation, Optimization, Program parsing	Cray Compiling Environment	Fortran	Machine	Closed
Misinformation, Optimization, Translation	GCC	C/C++	Binary/Assembly	Open
Invalid memory access, Misinformation, Optimization	GNU Fortran	Fortran	Machine	Open
Misinformation, Optimization	IBM Fortran	Fortran	Machine	Closed
Optimization	Intel C++	C++	Machine	Open
Invalid memory access, Misinformation, Optimization, Translation	Intel Fortran	Fortran	Machine	Open
Optimization	Java 7	Java	Java Bytecode	Closed
Optimization	Kotlin	Kotlin	Java Bytecode	Open
Optimization, Translation	LLVM	C/C++, C#, OpenCL, Ruby, Scala	Assembly	Open
Misinformation, Optimization	PGI	Fortran	Machine	Closed
Type	Solc	Solidity	Machine	Open
Misinformation, Optimization	Visual Studio x64	C++	Binary/DLL/Machine	Closed
Bit Arithmetic	WebAssembly	C/C++, Rust, C#, Kotlin, Go, Swift	Webassembly binary	Open
Misinformation	XCode 4	C/C++, Objective C/C++, Swift	Machine	Closed
Misinformation	Xilinx SDK	C++	Machine	Closed

Table 12. Benchmarks used in Peer-reviewed Publications

Index	Benchmark
P1	120 real compiler bugs (60 GCC bugs and 60 LLVM bugs), as well as 90 bugs collected from prior work (45 GCC bugs and 45 LLVM bugs).
P2	603 bugs (318 TVM bugs, 145 Glow bugs, and 140 nGraph bugs)
P3	No benchmark reported
P4	12% of the fixed miscompilation bugs for the Clang/LLVM C/C++ compiler
P5	GCC and LLVM
P6	320 typing-related bugs for four mainstream JVM languages, namely Java, Scala, Kotlin, and Groovy
P7	OpenCL systems
P8	No benchmark reported
P9	60 GCC bugs
P10	83 bugs (44 GCC and 39 LLVM bugs)
P11	21 OpenCL systems
P12	124 GCC bugs and 93 LLVM bugs
P13	No benchmark reported
P14	756 tests cases of K-Java and KSolidity
P15	8,771 GCC optimization bugs and 1,564 LLVM optimization bugs
P16	136 bugs from GCC- 4.8.5 and 81 bugs from Clang-3.6.1
P17	1,723 tool-chain bugs from LLVM
P18	8 GNU bugs and 23 LLVM bugs
P19	144,847 Simulink models
P20	45 GCC bugs and 45 LLVM bugs
P21	50 bugs in the Kotlin compiler
P22	112 bugs in three versions of ChakraCore, Javascript Core, and V8
P23	220 bugs in GCC, LLVM, and Intel C++ Compiler
P24	18 bugs in the Rust Compiler
P25	4 bugs in the P4 Compiler
P26	158 bugs in V8, ChakraCore, Javascript Core, SpiderMonkey, Rhino, Nashorn, Hermes, JerryScript, QuickJS, Graaljs

The papers related to defect categorization can be divided into two groups: (i) generic software systems: the defect taxonomies presented in the following papers, “Orthogonal Defect Classification: A Concept for In-process Measurements” [8], “Not All Bugs Are The Same: Understanding, Characterizing, and Classifying Bug Types” [4], “Bug

Table 13. Mapping of Defect Categories to Compilation Steps

Category	Compilation Phase	Code Construct
Bit arithmetic	Translation	Arithmetic Operations
Circular validation	Translation	Arithmetic Operations
Identifier resolution	Translation	Identifier/Objects
Integer equality	Translation	Arithmetic Operations
Linkage	Translation	Identifier/Objects
Loop induction	Translation	Identifier/Objects
Invalid Memory Access	Translation	Memory
Misinformation	Translation	Identifier/Objects
Optimization	Optimization	Identifier/Objects
Program parsing	Parsing	Identifier/Objects
Tensor	Translation	Identifier/Objects
Translation	Translation	Identifier/Objects
Type	Translation	Types

Table 14. Appearance of Defect Categories in Previously-studied Software Systems

Category	Previously-studied Software System
Bit arithmetic	Not reported for prior software system
Circular validation	Not reported for prior software system
Identifier resolution	Not reported for prior software system
Integer equality	IBM Proprietary Software [8]
Linkage	IBM Proprietary Software [8], NASA Software Projects [52], Service-oriented Web Systems [6]
Loop induction	Not reported for prior software system
Invalid Memory Access	Mozilla Projects [57], NASA Software Projects [52]
Misinformation	NASA Software Projects [52], Service-oriented Web Systems [6]
Optimization	NASA Software Projects [52]
Program parsing	IBM Proprietary Software [8]
Tensor	TensorFlow-based machine learning systems [66]
Translation	Deep learning systems [24]
Type	Deep learning systems [24]

characteristics in open source software" [57], and "Defect Categorization: Making Use of a Decade of Widely Varying Historical Data" [52] are applicable for generic software projects. Amongst these three publications, the three papers namely, "Orthogonal Defect Classification: A Concept for In-process Measurements" [8], "Bug Characteristics in open-source software" [58], and "Defect Categorization: Making Use of a Decade of Widely Varying Historical Data" [52] are seminal publications with high impact in the domain of software engineering research; and (ii) specialized software systems: the defect taxonomies presented in the following papers "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts" [42], "Taxonomy of Real Faults in Deep Learning Systems" [24], and "An Empirical Study on TensorFlow Program Bugs" [66] respectively, present defect categories for infrastructure as code, deep learning software, and Tensorflow. All of these software systems serve a unique purpose. Our hypothesis is that as these papers are recent and address relatively novel types of software, overlaps between our identified defect categories and existing categories in these papers can help us contextualize the novelty of compiler defects.

By considering publications from the above-mentioned groups we assume to synthesize existing reported defect categories, and then compare our identified compiler defect categories to that with existing defect categories for previously studied software systems. Our findings are reported in Table 14. The defect categories that have not been reported for prior software systems are bit arithmetic, circular validation, identifier resolution, and loop induction.

Mapping of Defect Categories with Compiler Components: We provide a mapping between each identified defect category and compiler components in Table 15. We observe the syntax analyzer and code generator respectively, to be mapped to 8 and 7 of the 13 defect categories. The intermediate representation generator is the least mapped component.

Table 15. Mapping Between Compiler Components and Defect Categories

Category	Compiler Component
Bit arithmetic	Semantic analyzer, Lexical analyzer, Syntax analyzer, Code generator
Circular Validation	Code generator
Identifier Resolution	Syntax analyzer
Integer equality	Lexical analyzer, Syntax analyzer
Linkage	Code generator
Loop Induction	Code optimizer, Intermediate representation generator, Semantic analyzer
Invalid Memory Access	Syntax analyzer, Code generator
Misinformation	Syntax analyzer, Semantic analyzer, Code generator
Optimization	Semantic analyzer, Code optimizer, Code generator
Program parsing	Lexical analyzer
Tensor	Semantic analyzer, Syntax analyzer
Translation	Syntax analyzer, Code generator
Type	Semantic analyzer, Syntax analyzer

Answer to RQ2: We identify 13 defect categories from our MLR of which bit arithmetic, circular validation, identifier resolution, and loop induction have not been reported for other software systems.

3.3 Answer to RQ3

We provide answers to *RQ3: What techniques have been reported in Internet artifacts and peer-reviewed publications to identify defects in compilers?* in this section.

3.3.1 Answer to RQ3: Defect Identification Techniques. Altogether, we identify 15 techniques used to identify defect categories that we describe below:

Deep learning: We observe deep learning algorithms to be used to identify defects in compilers. Using deep learning algorithms, programs are generated automatically, which are later executed to identify defects in compilers. For example, Cummins et al. [12] used deep learning to generate C programs to identify defects in GCC.

Reinforcement learning: We observe reinforcement learning to be used to find defects in compilers. In reinforcement learning, an agent is rewarded if the agent is executing steps toward the desired goal. Reinforcement learning was used by Chen et al. [7] to identify latent defects in GCC.

Tensor mutation: We observe tensor mutations to be used to identify defects in deep learning compilers. Shen et al. [54] studied defect characteristics in deep learning compilers and observed intermediate representation pre-processors to be the most defect prone. Using this observation, Shen et al. [54] constructed TVMFuzz that randomly mutates tensor types, tensor shapes, and tensor element values.

Optimization pattern synthesis: We observe researchers synthesizing patterns when a compiler performs optimizations to identify defects in compilers. Lim and Debray [32] mined and synthesized patterns in intermediate representation forms to identify defects in JIT compilers. Listing 18 shows an example of a program that is generated using optimization pattern synthesis [32]. The code snippet `a = i & -0;` is generated by (i) mutating a set of input programs, (ii) executing the mutated programs, and (iii) synthesizing an intermediate representation from the executions.

Differential testing: We observe differential testing to be used to identify defects in compilers. Differential testing applies the same input combinations to different variants of the same computer program and observes differences in

```
var a, b;
for (var i = 0; i < 100000; i++) {
    b = 1;
    a = i & -0; // Changed from '+' to '&'.
    b = a;
}
print(a === b);
gc();
print(a === b);
```

Listing 18. Javascript code snippet generated by optimization pattern synthesis [32].

the execution profile to detect unexpected behaviors in the program. Differential testing is used by Yang et al., where they constructed CSmith to identify defects in GCC. As another example, differential testing is used by Sun et al. [56] to find compiler defects for GCC. A variant of differential testing is the equivalence of modulo input, which was used by Chowdhury et al. [10] to identify defects in the Simulink compiler.

Markov Chains: We observe Monte Carlo Markov Chains (MCMCs) to be used to identify defects in compilers. Le et al. [31] constructed Athena, which uses MCMC to generate programs that are executed to identify defects in GCC and LLVM. Le et al. [31] used MCMC to find samples of program statements to generate programs.

Address Discrepancy Analysis: We observe address discrepancy analysis to be used to find defects in high-level synthesis (HLS) compilers. Fezzardi et al. [18] used address discrepancy analysis to identify invalid memory access defects in HLS compilers. As part of this analysis technique, Fezzardi et al. [18] uses HLS information to map software pointers with hardware memory access by constructing finite state machines.

Skeletal Program Enumeration: We observe skeletal program enumeration to be used to identify defects in compilers. Zhang et al. [65] observed that a computer program could be represented as a skeleton, i.e., a syntactic structure parameter by a collection of identifiers, for example, variables. Zhang et al. [65] applied partitioning to apply skeletal program enumeration to identify defects in GCC and Clang.

Semantic specifications: We observe skeletal semantic specifications to be used to identify defects in compilers. Schumi and Sun [51] used semantic specification where they generated structural operational semantic rules to generate programs to identify defects in the Java and the Solidity compiler.

Commercial static analysis tool usage: We observe practitioners use commercial static analysis tools to identify defects in compilers. For example, in a blog post, a practitioner mentioned how 'PVS Studio', a commercial static analysis tool was used to identify a invalid memory access defect in the GCC compiler.

Aspect preserving mutation: We observe Park et al. [39] to identify aspects, i.e., desirable properties in Javascript-like programs, and preserve aspects through stochastic mutation of the programs to identify defects for JavaScript compilers, such as the V8 JavaScript compiler.

Type-centric enumeration: We observe Stepanov et al. [55] to use type-centric enumeration to identify defects type-related defects for the Kotlin compiler. Type-centric enumeration is inspired by skeletal program enumeration, which leverages typed expression generation and type placeholder filling where generated expressions are mutated.

Constraint logic programming: We observe Dewey et al. [16] to use constraint logic programming to identify type-related defects for the Rust compiler. The goal is to generate well-typed Rust programs that can expose latent type-related defects in the Rust compiler. Dewey et al. [16] leveraged the Curry-Howard Correspondence where logical propositions correspond to types and programs correspond to proof terms.

Equivalence modulo input: We observe researchers use the concept of equivalence module input (EMI) to identify defects in compilers. EMI takes a computer program and a set of values as input and executes the program from which program profiles are extracted. Next, from the extracted program profiles EMI generates a set of program inputs by mutating the original input set so that the execution of the program is exactly the same as the original inputs.

User action: We observe regular user actions to lead to the discovery of defects in compilers. Unlike the above-mentioned techniques, for this category, users do not intentionally use a technique to identify defects in compilers. Instead, while using a compiler in a particular context, the defect in the compiler gets exposed.

Unlike all other reported techniques, for user action, no systematic technique is applied to discover a defect in the compiler. This category includes all actions performed by a compiler user when executing a computer program with a compiler. Let us consider the case of identifying a defect in the MingW64 component of GCC [64]. The user in this case was developing a model for ocean environments in order to approximate the health of fish stocks. The user was refactoring an existing implementation of the model to instantiate multiple models as threads. As part of this refactoring operation, the user identified a defect that resulted in erroneous calculations. Instead of receiving `1999.818926297566804`, the user received `1999.8189264475995515` with the refactored implementation. The erroneous calculation was attributed to a linkage defect where a function call was not linked to the implementation of `_fpreset`. The user further added: "*All in all I spent around 5 days chasing this bug through my code. I generated Gigabytes of log files and had to get down to the precision of 7.5 grains of sand on the planet Earth. The compiler missing a key function call turned out to be the cause of the issue. Many times, while trying to find the root cause I found myself questioning my ability to write code, diagnose bugs and remain sane. I'm glad I found an answer and have a way forward*" [64].

The identified 15 techniques can be divided into two groups: techniques identified from Internet artifacts and techniques identified from peer-reviewed publications. Two techniques namely, commercial static analysis tool usage and user action have been reported in Internet artifacts but not in peer-reviewed publications. The only technique that appears for both Internet artifacts and peer-reviewed publications is differential testing. Also, the techniques that we only obtain from peer-reviewed publications: address discrepancy analysis, aspect preserving mutation, constraint logic programming, deep learning, equivalence modulo input, Markov chains, optimization pattern synthesis, reinforcement learning, semantic specification, skeletal program enumeration, type-centric enumeration, and tensor mutation.

We provide a mapping between the applied technique and the corresponding Internet artifact in Table 16. We observe user action to be the most frequently applied technique to identify defects in compilers for Internet artifacts.

We also provide a mapping between the applied technique and the corresponding publication in Table 17. We observe differential testing to be the most frequently applied technique to identify defects in compilers for our set of peer-reviewed publications. P6, P15, and P17 use qualitative analysis to characterize reported defects and hence are not mapped to any technique in Table 17.

Table 16. Mapping Between Internet Artifacts and Techniques

Category	Artifact Index	Count
Differential testing	IA5	1
Commercial static analysis tool	IA16	1
User action	IA1, IA2, IA3, IA4, IA6, IA7, IA8, IA9, IA10, IA11, IA12, IA13, IA14, IA15, IA17, IA18, IA19, IA20, IA21, IA22, IA23, IA24, IA25, IA26, IA27, IA28, IA29, IA30, IA31, IA32	30

Table 17. Mapping Between Publications and Techniques

Category	Publication Index	Count
Address discrepancy analysis	P13	1
Deep learning	P7, P26	2
Differential testing	P4, P5, P8, P9 , P11, P12, P14, P18, P19, P23, P25	11
Equivalence modulo input	P22	1
Markov chains	P10, P20	2
Optimization pattern synthesis	P3	1
Reinforcement learning	P1	1
Semantic specification	P19	1
Skeletal program enumeration	P16, P20	2
Tensor mutation	P2	1
Aspect preserving mutation	P22	1
Type-centric enumeration	P21	1
Constraint logic programming	P24	1

Table 18. Mapping Between Defect Categories and Techniques

Technique	Defect Category
Address discrepancy analysis	Bit arithmetic
Deep learning	Program Parsing, Type
Differential testing	Translation, Loop Induction, Misinformation, Optimization, Invalid Memory Access, Program parsing, Type
Equivalence modulo input (EMI)	Invalid Memory Access, Optimization, Transalation
Markov chains	Misinformation, Optimization, Bit arithmetic
Optimization pattern synthesis	Optimization
Reinforcement learning	Bit arithmetic, Optimization
Semantic specification	Misinformation
Skeletal program enumeration	Integer equality, Translation, Bit arithmetic
Commercial static analysis tool	Misinformation
Tensor mutation	Tensor, Type
User action	Bit Arithmetic, Invalid Memory Access, Misinformation, Optimization, Program parsing, Translation, Type
Aspect preserving mutation	Type
Type-centric enumeration	Type
Constraint logic programming	Type

We further provide details on techniques used to identify certain defect categories in Table 18. We observe two techniques to be dominant: differential testing and user action. Unlike user action, differential testing is systematic and can be used to generate programs to automatically find defects in compilers. We also observed no one technique is enough to identify all identified defect categories in compilers.

3.3.2 Answer to RQ3: Challenges Addressed by Identified Techniques. We describe the challenges that are addressed by the 15 techniques. We identify five categories of challenges that we describe below. A mapping between each identified technique and the challenge it identifies is listed in Table 19.

Table 19. Mapping Between Techniques and Addressed Challenges

Technique	Addressed Challenge
Address discrepancy analysis	Optimized memory localization for HLS
Deep learning	Automated generation of test programs
Differential testing	Automated generation of test programs
Equivalence modulo input (EMI)	Automated generation of inputs for test programs
Markov chains	Automated generation of test programs
Optimization pattern synthesis	Automated generation of test programs
Reinforcement learning	Automated generation of test programs
Semantic specification	Automated generation of test programs
Skeletal program enumeration	Automated generation of test programs
Commercial static analysis tool usage	N/A
Tensor mutation	Tensor attribute mining
User action	N/A
Aspect preserving mutation	Aspect Pre-condition Analysis

Optimized memory localization for HLS: This category corresponds to the unique challenge of identifying the memory location to expose memory-related defects in the HLS compiler. To address this unique challenge authors of P13 used address discrepancy analysis.

Automated generation of inputs for test programs: This category corresponds to the challenge of generating input data for existing computer programs that are used to trace compiler executions. To address this challenge researchers have used equivalence modulo input.

Automated generation of test programs: This category corresponds to the challenge of generating computer programs in a certain programming language so that these programs can identify latent defects in compilers. This category is different from the automated generation of inputs, as the category only considers the generation of test programs and not the generation of inputs for existing programs. From our analysis we find generating test programs accurately and effectively to find compiler bugs is challenging. To address this challenge researchers have used a wide range of techniques, namely, deep learning, differential testing, reinforcement learning, Markov chains, optimization pattern synthesis, semantic specification, and skeletal program enumeration.

Tensor attribute mining: This category corresponds to the challenge of transforming deep learning programs into adequate forms so that deep learning compilers can be fuzzed. To address this challenge, authors of P2 apply tensor mutation using the following steps: (i) construction of directed graphs based on API calls, and (ii) select random subgraphs from step (i) and mutate the graphs for tensor type, tensor shape, and primitive tensor values.

Aspect Pre-condition Analysis: This category corresponds to the challenge of the pre-condition necessary to identify and model an aspect, i.e., a desirable property in Javascript programs. We observe Park et al. [39] to address this challenge with aspect-preserving mutation.

In Table 19 'N/A' corresponds to a technique not addressing challenges as reported in an Internet artifact. The two techniques commercial static analysis tool usage and user action are reported in Internet artifacts that have not mentioned any challenge the mentioned technique it addresses.

We also report the temporal trends of the studied challenges reported in Figure 8. We observe the most frequently studied challenge is the automated generation of test programs ('GENERATE-PROGRAM' in Figure 8). The first publication related to the automated generation of test programs in our set was published in 2011. We observe Tensor attribute mining ('TENSOR-MINING') to be a relatively recent topic of interest amongst researchers.

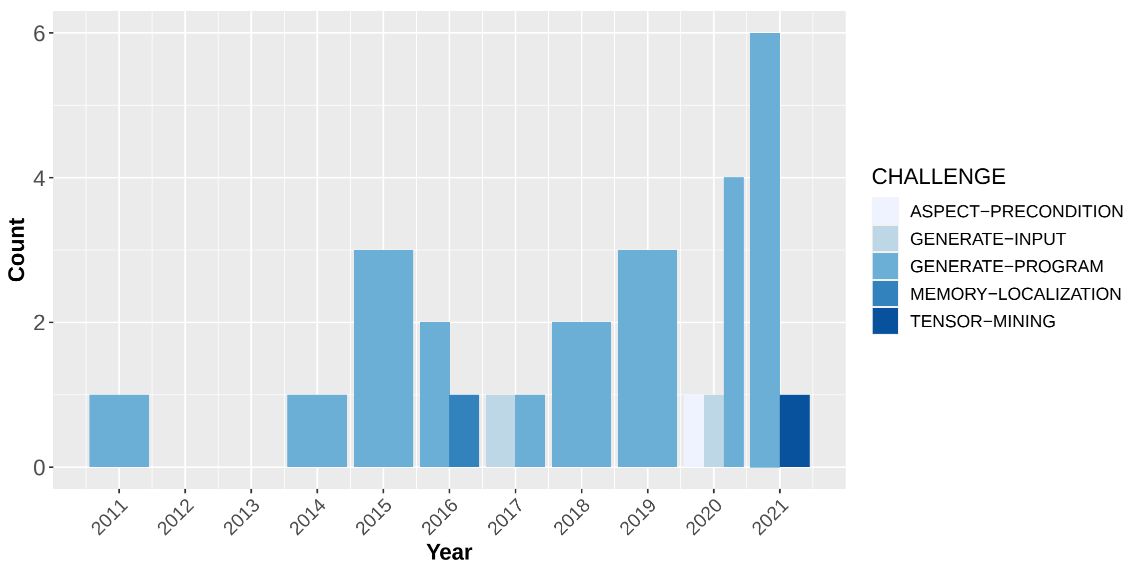

Fig. 8. Temporal trends of studied challenges in our set of 26 peer-reviewed publications.

Answer to RQ3: We identify 15 techniques to identify defects in compilers. The most frequently used technique amongst publications is differential testing, whereas the most frequently used technique in Internet artifacts is user action.

4 DISCUSSION

We discuss the findings of our MLR paper as follows:

Usefulness of Differential Testing From Table 17, we observe differential testing to be the most frequently used technique to identify defects in compilers. 11 papers use differential testing, and each of these 11 papers has reported differential testing and its variants to be effective in identifying defects. Despite documented benefits reported in publications, we observe differential testing under-reported in Internet artifacts. Only one artifact reported this technique to be used to find defects in compilers. These observations imply that for the systematic identification of defects in compilers, practitioners can rely on differential testing, as there is documented evidence of the effectiveness of differential testing for finding defects in a diverse set of compilers, such as GCC, LLVM, and Simulink.

Studied Compilers - Differences and Similarities Between Publications and Internet Artifacts: From Section 3.1, we notice both differences and similarities with respect to the studied compiler in our MLR. In both Internet artifacts and publications, GCC and LLVM are well-investigated compilers. However, in the set of Internet artifacts, we observe the following compilers to be investigated, which are absent in publications: Arduino SDK, GNU Fortran, Intel Fortran, Java 7 Compiler, PGI Fortran, Code Composer Studio, Cray Compiling Environment, IBM Fortran, Kotlin, MingW64, Visual Studio x64, WebAssembly, XCode, and Xilinix SDK. One possible explanation is practitioners report defects for compilers that they use to perform their professional responsibilities. On the other hand, in the case of peer-reviewed publications defects are reported as part of the scientific discovery that may not overlap with compilers that practitioners use in niche domains.

The results of RQ1 have implications for researchers. Results of RQ1 reveal that there is a wide range of compilers that practitioners use and include defects. The implication of this finding is that researchers can apply existing defect finding techniques for compilers that practitioners use but have not thoroughly investigated by researchers. For example, for four Fortran-related compilers, namely, IBM Fortran, Intel Fortran, PGI Fortran, and GNU Fortran practitioners, have reported defects. Researchers can investigate if techniques applicable for GCC are applicable to these Fortran-related compilers. According to enlyft [9], 13,031 companies use Fortran. Defect identification techniques for Fortran compilers can help practitioners who use Fortran for commercial and scientific purposes [26]. As another example, researchers can investigate if existing defect identification techniques can be applied to Kotlin, which is used by 60% of professional Android app developers [10]. Our hypothesis is existing defect identification techniques used in existing research may not work for unexplored compilers, such as Kotlin and Fortran.

> *Implication#1: By comparing the studied compilers between Internet artifacts and peer-reviewed publications, we observe Arduino SDK, GNU Fortran, Intel Fortran, Java 7 Compiler, PGI Fortran, Code Composer Studio, Cray Compiling Environment, IBM Fortran, MingW64, Visual Studio x64, WebAssembly, XCode, and Xilinix SDK not to be studied peer-reviewed publications. We advocate researchers apply existing and novel defect identification techniques for compilers that practitioners use but have not thoroughly investigated by researchers, such as Fortran compilers.*

Identified Defect Categories of Compilers: From Section 3.2, we observe specific defect categories to be unique to compilers that do not appear for other software systems. These defect categories are bit arithmetic, circular validation, identifier resolution, and loop induction. This observation implies that defects in compilers have unique characteristics and thus require systematic investigation specific to compilers. Defect categories that appear for compilers and other software systems are integer equality, linkage, invalid memory access, misinformation, optimization, program parsing, tensor, translation, and type.

We identify the following defect categories that we observe in peer-reviewed publications but not in Internet artifacts: circular validation, identifier resolution, integer equality, linkage, loop induction, and tensor. All six defect categories observed for Internet artifacts are documented in peer-reviewed publications. One possible explanation is that researchers who author peer-reviewed publications systematically apply a set of techniques in order to identify defects in compilers. Unlike software practitioners who use compilers, researchers of our studied peer-reviewed publications are experts in the domain of compiler testing. Their expertise, as demonstrated through their research activities, might have helped in yielding the defect categories not reported in peer-reviewed publications.

> *Implication#2: Practitioners might not systematically apply techniques to find defects in the compilers they use. Researchers should proactively engage in defect identification research in compilers that have relevance for practitioners and aid in making the software supply chain resilient.*

[9] https://enlyft.com/tech/products/fortran
[10] https://developer.android.com/kotlin

Defect Identification Techniques: From Section 3.3, we observe three techniques that have been reported in Internet artifacts. The count of defect categories reported in Internet artifacts is also lower than that of peer-reviewed publications. One possible explanation is practitioners are aware of defect identification techniques used by researchers, but such analyses are not reported publicly, especially for compilers that are closed source, such as Code Composer Studio. Another possible explanation is that practitioners are more users of compilers who may not have the necessary expertise to perform compiler testing. As a result, practitioners only use a handful set of techniques to identify defects in compilers. Such explanation can partially be substantiated by findings reported in Table 16. We observe user action to be the most frequently reported technique amongst Internet artifacts. User action is the technique when a compiler user uses a compiler to perform a task, but while performing the task, a defect in the compiler is exposed. User action is not a systematic compiler testing technique, which may not yield all possible defect categories. Our explanation is subject to empirical substantiation, which researchers can investigate further.

Implication#3: Researchers can systematically investigate if practitioners are aware of defect identification techniques for compilers through interviews and/or survey analysis. Based on the conducted research, researchers can further investigate how defect identification techniques that are common in peer-reviewed research can be transitioned to industry. Existing research [3, 33] related to software quality assurance could be of interest to researchers in this regard.

Latent Defects in Infrastructure Orchestrators: Modern day computing infrastructure is managed with domain-specific languages called infrastructure as code (IaC) languages [43, 44]. IaC is the practice of automatically managing computing infrastructure at scale with dedicated programming languages [43, 44]. Languages used for IaC are examples of domain specific languages, which are different from general purpose programming languages, such as C and Java [45]. From our results reported in Section 3.2, we observe a lack of research related defects in IaC orchestrators, i.e., software tools that parse and compile IaC software artifacts to manage large-scale computing infrastructure. As these languages are pivotal in automated provisioning of computing infrastructure, the underlying compilers that process and translate IaC scripts need to be robust and resilient. To that end, we propose the following research directions: (i) gain an understanding of defects in IaC orchestrators through categorization; (ii) discover latent defects in IaC orchestrators with established techniques, such as differential testing and fuzzing; and (iii) formal verification of orchestrator components with theorem proving.

Implication#4: As IaC is an emerging domain, as part of future work, researchers can investigate techniques to identify latent defects in IaC orchestrators, i.e., software tools that parse and compiler IaC scripts.

5 THREATS TO VALIDITY

We discuss the limitations of our paper as follows:

Conclusion Validity: Our application of inclusion and exclusion criteria is susceptible to rater bias, which can limit the sets of Internet artifacts and peer-reviewed publications that we have used in our MLR. We mitigate this limitation by using two raters and a resolver who resolved the disagreements between the two raters. Our approach to deriving defect categories is susceptible to rater bias, as these categories are derived by the third author. We mitigate this limitation by performing rater verification.

We acknowledge that our list of keywords to search Internet artifacts and peer-reviewed publications might not be comprehensive. We mitigate this limitation by using a quasi-gold set. We also acknowledge that our results are limited to the quality of the Internet artifacts and peer-reviewed publications, which we mitigate by conducting a quality analysis.

Construct Validity: Our MLR involves the application of qualitative analysis conducted by the third author, which we use to answer RQ1, RQ2, and RQ3. The third author is a Ph.D. student with two years of experience in professional software engineering. Such experience of the rater makes the conducted qualitative analysis susceptible to mono-method bias, i.e., the phenomenon of rater expectation to influence the outcomes of the qualitative analysis. We mitigate this limitation by performing rater verification and allocating another rater.

External Validity: Our answers to RQ1, RQ2, and RQ3 are limited to the sets of Internet artifacts and publications that we collected. With the evolution of time, the count of Internet artifacts and publications related to compiler defects can grow. Therefore, a potential future review of Internet artifacts and publications related to compiler defects can identify defect categories that are not included in our paper.

6 RELATED WORK

Our paper is closely related to MLRs that have been conducted in the domain of automated software engineering. Myrbakken and Colomo-Palacios [37] performed an MLR to identify the benefits and challenges of adopting security in development and operations (DevOps) with two peer-reviewed publications and 50 Internet artifacts. Sanchez-Gordon et al. [49] reported growing interest in DevOps adoption for developing e-learning systems with their MLR. Garousi and Mantyla [23] performed an MLR study and provided a checklist of practical advice for practitioners for better software test automation. In another work, Garousi et al. [20] performed an MLR with 130 peer-reviewed publications and 51 Internet artifacts and reported 58 software test maturity models, five driving factors, three benefits, and eight challenges for conducting successful test maturity assessment and test process improvement.

The examples mentioned earlier showcase the community's interest in using MLRs to derive novel and actionable insights for practitioners related to software engineering.

Our paper is also related to prior research on defect categories for software. In 1992, Chillarege et al. [9] proposed Orthogonal Defect Classification (ODC) that included eight defect categories. Categories proposed by Chillarege et al. [9] were used by Cinque et al. [11] to categorize defects for air traffic control software. Later in 2008, Seaman et al. [52] extended ODC to derive 7 categories of requirements defects and 7 categories of test plan defects. Use of existing defect categorization frameworks, such as ODC and Seaman et al. [52]'s work, may be inadequate for compilers, as observed in Table 14.

Researchers have also categorized defects for domain-specific software systems. For example, Humbatova et al. [24] mined GitHub issues and Stack Overflow posts to derive a fault taxonomy for software projects that use deep learning. Rahman et al. [42] used open coding with commits to derive defect categories for Puppet scripts.

We observe a lack of research related to compiler defect categorization. We have used an MLR to characterize defects in compilers, which can help practitioners to improve the quality of compilers.

7 CONCLUSION

Compilers play a pivotal role in software development as they compile code into a format that the processor can execute. Hence, defects in compilers can be disruptive for software developers and thus needs to be systematically identified. We have conducted an MLR to help practitioners and researchers identify defects in compilers. From our MLR, we identify 13 defect categories. We also identify 15 techniques, amongst which differential testing is the most frequently used technique in the 26 publications used for our MLR. However, we also observe that one technique is not enough to identify all defect categories reported in publications and Internet artifacts. Based on our findings, we recommend the systematic application of techniques listed in peer-reviewed publications to identify defects in compilers. These techniques can automatically generate computer programs, which in turn can expose latent defects in compilers.

ACKNOWLEDGMENTS

We thank the PASER group at Auburn University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2310179, and Award # 2312321.

REFERENCES

[1] 2010. Can compiler optimization introduce bugs? https://stackoverflow.com/questions/2722302/can-compiler-optimization-introduce-bugs

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[3] Jeffrey C. Carver, Oscar Dieste, Nicholas A. Kraft, David Lo, and Thomas Zimmermann. 2016. How Practitioners Perceive the Relevance of ESEM Research. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 56, 10 pages. https://doi.org/10.1145/2961111.2962597

[4] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types. *J. Syst. Softw.* 152, C (jun 2019), 165–181. https://doi.org/10.1016/j.jss.2019.03.002

[5] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[6] KS Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. 2007. A fault taxonomy for web service composition. In *International Conference on Service-Oriented Computing*. Springer, 363–375.

[7] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.

[8] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. 1992. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering* 18, 11 (1992), 943–956. https://doi.org/10.1109/32.177364

[9] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. 1992. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering* 18, 11 (Nov 1992), 943–956. https://doi.org/10.1109/32.177364

[10] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. 2020. SLEMI: Finding Simulink compiler bugs through equivalence modulo input (EMI). In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 1–4.

[11] Marcelo Cinque, Dominico Cotroneo, Raffaele D. Corte, and Antonio Pecchia. 2014. Assessing Direct Monitoring Techniques to Analyze Failures of Critical Industrial Systems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 212–222. https://doi.org/10.1109/ISSRE.2014.30

[12] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105.

[13] Philippe Daouadi. 2019. Miscompilation: clang generates a weak symbol for a generic lambda that depends on a lambda in a static function. https://bugs.llvm.org/show_bug.cgi?id=44368

[14] Henry Davie. 1979. Assemblers and loaders: DW Barron, Macdonald and Janes Computer Monographs (1978)£ 3.95 102pp.

[15] developernation. 2021. State of the Developer Report. https://www.developernation.net/developer-reports/de20

[16] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 482–493. https://doi.org/10.1109/ASE.2015.65

[17] Robert J Ellison, John B Goodenough, Charles B Weinstock, and Carol Woody. 2010. *Evaluating and mitigating software supply chain security risks*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

[18] Pietro Fezzardi and Fabrizio Ferrandi. 2016. Automated bug detection for pointers and memory accesses in High-Level Synthesis compilers. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–9.

[19] Friesen. 2022. The setup. https://www.nhn.ou.edu/~friesen/cray_compiler_bug.html

[20] Vahid Garousi, Michael Felderer, and Tuna Hacaloğlu. 2017. Software test maturity assessment and test process improvement: A multivocal literature review. *Information and Software Technology* 85 (2017), 16–42.

[21] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2016. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th international conference on evaluation and assessment in software engineering*. 1–6.

[22] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101–121.

[23] Vahid Garousi and Mika V Mäntylä. 2016. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology* 76 (2016), 92–117.

[24] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[25] Chris Fallin iximeow. 2021. Defense in depth: stopping a Wasm compiler bug before it became a problem. https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem

[26] Upulee Kanewala and James M Bieman. 2014. Testing scientific software: A systematic literature review. *Information and software technology* 56, 10 (2014), 1219–1232.

[27] Barbara Kitchenham, Dag IK Sjøberg, Tore Dybå, O Pearl Brereton, David Budgen, Martin Höst, and Per Runeson. 2012. Trends in the Quality of Human-Centric Software Engineering Experiments–A Quasi-Experiment. *IEEE Transactions on Software Engineering* 39, 7 (2012), 1002–1017.

[28] Barbara Kitchenham, Dag IK Sjøberg, Tore Dybå, O Pearl Brereton, David Budgen, Martin Höst, and Per Runeson. 2012. Trends in the Quality of Human-Centric Software Engineering Experiments–A Quasi-Experiment. *IEEE Transactions on Software Engineering* 39, 7 (2012), 1002–1017.

[29] Marco Kuhrmann, Daniel Méndez Fernández, and Maya Daneva. 2017. On the pragmatic design of literature studies in software engineering: an experience-based guideline. *Empirical software engineering* 22, 6 (2017), 2852–2891.

[30] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

[31] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.

[32] HeuiChan Lim and Saumya Debray. 2021. Automated bug localization in JIT compilers. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 153–164.

[33] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. How Practitioners Perceive the Relevance of Software Engineering Research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 415–425. https://doi.org/10.1145/2786805.2786809

[34] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 460–472. https://doi.org/10.1109/ICSE43902.2021.00051

[35] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[36] Robert Morgan. 1998. *Building an optimizing compiler*. Digital Press.

[37] Håvard Myrbakken and Ricardo Colomo-Palacios. 2017. DevSecOps: a multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination*. Springer, 17–29.

[38] George C Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices* 33, 5 (1998), 333–344.

[39] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[40] Akond Rahman. 2023. Dataset for Paper. https://tigermailauburn-my.sharepoint.com/:f:/g/personal/azr0154_auburn_edu/Eh9ifbPqv8dLjZ5H9bmSUsUBs23QaCCs785ot-Jv6ewNaQ?e=vOqIju. [Online; accessed 20-August-2023].

[41] Akond Rahman, Farhat Lamia Barsha, and Patrick Morrison. 2021. Shhh!: 12 practices for secret management in infrastructure as code. In *2021 IEEE Secure Development Conference (SecDev)*. IEEE, 56–62.

[42] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3377811.3380409 pre-print: https://akondrahman.github.io/papers/icse20_acid.pdf.

[43] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2018. A systematic mapping study of infrastructure as code research. *Information and Software Technology* (2018). https://doi.org/10.1016/j.infsof.2018.12.004

[44] A. Rahman and C. Parnin. 2023. Detecting and Characterizing Propagation of Security Weaknesses in Puppet-Based Infrastructure Management. *IEEE Transactions on Software Engineering* 49, 06 (June 2023), 3536–3553. https://doi.org/10.1109/TSE.2023.3265962

[45] Akond Rahman and Laurie Williams. 2018. Characterizing Defective Configuration Scripts Used for Continuous Deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 34–45. https://doi.org/10.1109/ICST.2018.00014

[46] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. 2015. Synthesizing Continuous Deployment Practices Used in Software Development. In *Proceedings of the 2015 Agile Conference (AGILE '15)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/Agile.2015.12

[47] Austen Rainer. 2017. Using Argumentation Theory to Analyse Software Practitioners Defeasible Evidence, Inference and Belief. *Inf. Softw. Technol.* 87, C (jul 2017), 62–80. https://doi.org/10.1016/j.infsof.2017.01.011

[48] Johnny Saldana. 2015. *The Coding Manual for Qualitative Researchers*. SAGE.

[49] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. 2018. A multivocal literature review on the use of DevOps for e-learning systems. In *Proceedings of the Sixth International Conference on Technological Ecosystems for Enhancing Multiculturality*. 883–888.

[50] Sean Patrick Santos. 2014. Fortran derived type (mostly structure constructor) bugs. https://forums.developer.nvidia.com/t/fortran-derived-type-mostly-structure-constructor-bugs/134017

[51] Richard Schumi and Jun Sun. 2021. SpecTest: Specification-Based Compiler Testing. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 269–291.

[52] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect Categorization: Making Use of a Decade of Widely Varying Historical Data. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany) *(ESEM '08)*. Association for Computing Machinery, New York, NY, USA, 149–157. https://doi.org/10.1145/1414004.1414030

[53] M. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman. 2020. XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. In *2020 IEEE Secure Development (SecDev)*. IEEE Computer Society, Los Alamitos, CA, USA, 58–64. https://doi.org/10.1109/SecDev45635.2020.00025

[54] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 968–980.

[55] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. 2021. Type-Centric Kotlin Compiler Fuzzing: Preserving Test Program Correctness by Preserving Types. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 318–328. https://doi.org/10.1109/ICST49551.2021.00044

[56] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 203–213.

[57] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.

[58] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19 (2014), 1665–1705.

[59] Akond Ashfaque Ur Rahman and Laurie Williams. 2016. Software Security in DevOps: Synthesizing Practitioners' Perceptions and Practices. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery* (Austin, Texas) *(CSED '16)*. ACM, New York, NY, USA, 70–76. https://doi.org/10.1145/2896941.2896946

[60] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 1–10.

[61] Xiaoyuan Xie, Haolin Yang, Qiang He, and Lin Chen. 2021. Towards Understanding Tool-chain Bugs in the LLVM Compiler Infrastructure. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–11.

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[63] Bob Yantosca. 2010. Segmentation faults. http://wiki.seas.harvard.edu/geos-chem/index.php/Segmentation_faults

[64] Zaita. 2021. The weirdest compiler bug. https://blog.zaita.com/mingw64-compiler-bug/. [Online; accessed 20-Sep-2022].

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

[66] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[67] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884.