Noname manuscript No.
(will be inserted by the editor)

Come for Syntax, Stay for Speed, Understand Defects: An Empirical Study of Defects in Julia Programs

Akond Rahman · Dibyendu Brinto Bose · Raunak Shakya · Rahul Pandita

Received: date / Accepted: date

Abstract Julia has emerged as a popular programming language to develop scientific software, in part due to its flexible syntax akin to scripting languages while retaining the execution speed of a compiled language. Similar to any programming language, Julia programs are susceptible to defects. However, a systematic characterization of defects in Julia programs remains under-explored. A systematic analysis of defects in Julia programs will act as a starting point for researchers and toolsmiths in building developer tools to improve the quality of Julia programs. To this end, we conduct an empirical study with 742 defects that appear in Julia programs by mining 30,494 commits and 3,038 issue reports collected from 112 open-source Julia projects. From our empirical analysis, we identify 9 defect categories and 7 defect symptoms. We observe certain defect categories to be Julia-specific, e.g., type instability and world age defects. We also survey 52 developers to rank the identified categories based on perceived severity. Based on our empirical analysis, we provide specific recommendations for researchers and toolsmiths.

Keywords categorization · defects · empirical study · julia · quality

Akond Rahman
Auburn University
E-mail: akond@auburn.edu

Dibyendu Brinto Bose
Virginia Tech
E-mail: brintodibyendu@gmail.com

Raunak Shakya
Mineral Worths
E-mail: rkshakya99@gmail.com

Rahul Pandita
GitHub
E-mail: rahulpandita@github.com

1 Introduction

Scientific software is typically developed in scripting languages, such as `Python` and `R` due to ease in iterative and exploratory development (Bezanson et al., 2018). However, to increase program execution speed, scientific software developed in such scripting languages is often migrated to compiled languages, such as `C` and `Fortran` as they provide more predictable mapping to underlying hardware that in turn results in optimal speed of execution (Bezanson et al., 2018). These migrations are accompanied with a development and maintenance overhead (Bezanson et al., 2018). `Julia` was designed to address this issue by providing programming syntax similar to scripting languages, without sacrificing program execution speed (Jul, 2019; Bezanson et al., 2018); often colloquially referred to as *"come for the syntax, stay for the speed"*.

Ever since its inception in 2012 Julia has experienced a steady increase in popularity (jul, 2022) as more scientific software developers are migrating from scripting languages, such as Python to Julia (jul, 2020c). According to a survey of Stack Overflow users in 2020, Julia is one of the "*top-10 most loved programming languages*" by developers (jul, 2020b). As of March 2022, Julia has been downloaded more than 34.8 million times (jul, 2022). Practitioners typically use Julia for large-scale scientific data analysis, e.g., Julia was used in Celeste (Jul, 2022a; jul, 2017), a software used in astronomy research. The use of Julia yielded a performance improvement by a factor of 1,000 for Celeste, compared to prior implementation (Jul, 2022a).

Despite reported benefits, Julia programs are susceptible to defects similar to other software systems. For instance, consider Listing 1, where we present an example of a defect downloaded from an OSS repository (jayschwa, 2014). The defect is due to incorrect `type` usage while returning type predicate of generated functions. Because of incorrect type usage, a crash occurred. The defect was repaired by using the correct type (`VegaMarkFrom`). Listing 1 shows an example of how a incorrect type usage can result in defects, and therefore, needs to be avoided.

Due to relative nascence of the ecosystem, a systematic investigation of defects in Julia programs is an under-explored area. Such investigation will be beneficial for the Julia community as such a study has the potential to yield insights on why defects in Julia programs appear, and derive actionable recommendations to mitigate such defects. Specifically, these recommendations will benefit (i) *researchers*: in developing an understanding of the nature of defects in Julia programs, and (ii) *toolsmiths*: to construct tools to improve the quality of Julia programs.

Contributions: We list our contributions as the following:

- A list of 9 defect categories for Julia programs;
- A list of 7 symptoms for defects in Julia programs;
- An evaluation of how frequently identified defect categories and symptoms occur; and
- An evaluation of how developers perceive identified defect categories.

```
(:name, String, nothing),
(:description, String, nothing),
- (:from, Dict{Any, Any}, nothing),
+ (:from, VegaMarkFrom, nothing),
(:properties, VegaMarkProperties, nothing),
(:key, String, nothing),
```

Listing 1: An example of a defect in a Julia program.

We organize the rest of the paper as follows: we list our research questions in Section 2. We provide background and related work in Section 3. We provide the methodology in Section 4. We answer RQ1, RQ2, and RQ3 in Sections 5, 6, and 7, respectively. We discuss our findings in Section 8. We provide the limitations of the paper in Section 9, and conclude the paper in Section 10.

2 Research Questions

In our empirical study, we answer the following research questions:

- ***RQ1 [Categorization]****: What categories of defects exist in Julia programs?*
- ***RQ2 [Symptoms]****: What are the symptoms of defects in Julia programs?*
- ***RQ3 [Perception]****: How do developers perceive the identified defect categories for Julia programs?*

By answering these research questions we will accomplish the goal of helping researchers and toolsmiths. Through this empirical study, we aim to help researchers by giving them a taxonomy of defects, showcasing exemplars for each defect category, and reporting which of the identified defect categories are applicable for other software systems. We also aim to help toolsmiths, who will develop tools for practitioners who use the Julia programming language. We aim to highlight the areas that need attention from toolsmiths so that practitioners get the support they need.

Two factors motivate us the conduct our empirical study. *First*, Julia is perceived to solve the 'two language problem', which allows practitioners to write computer programs that have good performance but can be written in scripting manner. This requires designing programming language features that are unique to Julia. Our hypothesis is that Julia's unique features can lead us to discovery of defect categories unique to Julia, and not observant other software systems. Results reported in Table 4 showcases that certain defect categories are unique to Julia programs. *Second*, our empirical study should adopt a methodology generic enough that so that can still identify defects unique to Julia but the adopted methodology can be applied other emerging programming languages. Along with identifying unique defect categories, from

Table 4 we also find defect categories that are applicable for other software systems, such as deep learning projects (Humbatova et al., 2020) and Puppet manifests (Rahman et al., 2020).

Furthermore, Julia is gaining in prominence amongst practitioners from academia, government, and industry typically use Julia for large-scale scientific data analysis (julia, 2021). For example: (i) the Los Alamos National Laboratory uses Julia for optimization of critical infrastructure to mitigate extreme events related to electricity delivery (Jul, 2022b); and (ii) the U.S. Federal Aviation Administration uses Julia to model airborne collision avoidance (Jul, 2022c). A defect categorization study can aid practitioners from all of these diverse domains.

We conduct an empirical study with 742 defects that appear in Julia programs by mining 30,494 commits and 3,038 issue reports from 112 open-source software (OSS) Julia projects. Using open coding (Saldaña, 2015), we derive defect categories and defect symptoms for Julia programs. We also survey 52 developers to analyze developer perception related to identified defect categories, as well as the perceived frequency and severity of the defect categories. Datasets used in our paper are publicly available online (Rahman, 2022).

Our empirical study is an example of a mixed methods approach (Easterbrook et al., 2008) where quantitative and qualitative data sources are used and investigated to get diverse perspectives on the conducted research. To that end, we have derived defect categories and symptoms using qualitative analysis, and also conducted survey to quantitively determine the frequency and severity of identified defect categories. The implications of identifying defect categories is related to generating research in an emerging domain, as well as provide recommendations for toolsmiths on what tools could be beneficial to Julia users. The implications of identifying defect symptoms is that it showcases what are the consequences of identified defect categories. For example, from Section 5 we have identified that certain defect categories can lead to incorrect calculations. Incorrect calculations are particularly relevant for the Julia user base, as practitioners use Julia to develop scientific software. Our survey analysis provides insight on what defect categories are perceived as severe. For example, we observe 65% of practitioners to identify type-related defects as severe or most severe. Further, we can compare and contrast the practitioner perceptions and the results reported in Section 7.

3 Background and Related Work

We provide necessary background information related to scientific software, the Julia programming language, and discuss related work.

3.1 Background

We provide background information as follows:

3.1.1 Background on Scientific Software

According to Carver (2009) scientific software is software that is used to investigate "*complex scientific problems*". Furthermore, according to Carver (2009) there are differences between scientific software and non-scientific software, such as

- Scientific software is dedicated to explore unknown science, which makes it difficult, if not impossible to derive a concrete set of requirements beforehand.
- Successful scientific software often revolves around it optimization to the machine architecture. Such pursuit related to optimization can incur more efforts than efforts necessary for software implementation.
- Execution of scientific software often requires powerful computing resources.

These differences were acknowledged by Howison and Herbsleb (2011) who stated for scientific software "*what is needed, as we see it, is to understand scientific software as an independent production system*".

3.1.2 Background on the Julia Programming Language

While conducting large-scale scientific experiments, performance is pivotal for researchers so that they can obtain scientific experiment results in a timely manner. Julia allows practitioners to develop programs in a scripting manner, and also at the same time provide utilities to write performant programs. We conduct this empirical study to understand the defects that occur in Julia programs.

Julia is designed to provide programming syntax similar to that of scripting languages, with the program execution speed of compiled languages, which have low-level memory access (Jul, 2019; Bezanson et al., 2018). According to Perkel (2019), "*Julia circumvents that two-language problem because it runs like C, but reads like Python*". Julia supports just-in-time compilation, multiple dispatch, annotations, and meta-programming (Bezanson et al., 2018; Jul, 2022d). All of Julia's types are organized hierarchically. '`Any`' is at the top of the hierarchy, and has 221 immediate subtypes, e.g., '`BigFloat`', '`BigInt`', '`Complex`' etc. (Jul, 2022d; Poulding and Feldt, 2017).

Listing 2 is an instance of Julia program. '`__precompile__()`' signals the Julia compiler to enable pre-compilation for the '`Example`' module. With pre-compilation, the Julia compiler loads the '`Sample.jl`' dependency once, and stores it in a cache. In the future, every time the '`Example`' is executed, contents of '`Sample.jl`' will be retrieved from the cache instead of loading '`Sample.jl`' from the persistence.

3.2 Related Work

This work is related to prior research that have investigated Julia programs and defect categorization:

```
__precompile__()
module Example
  # Include a package dependency
  include("Sample.jl")
  println("Hello World")
  # Function to multiply and add two values
  function mul_and_add(a, b)
     m = a*b
     a = a+b
     m, a
  end
```

Listing 2: An annotated example of a Julia program.

3.2.1 Research Related to Julia

Since its inception in 2012, Julia has garnered tremendous interest amongst researchers. Quality issues in Julia programs and the Julia compiler have been investigated, for example, Paulding and Feldt (2017) applied random testing to find defects in 9 functions provided by the Julia compiler. Churavy (2019) constructed a debugging tool called 'Cthulhu' that uses static and dynamic analysis to help developers find defects in array abstractions. Nardelli et al. (2018) used formal specification to verify the correctness of the Julia compiler's subtype system. Productivity and performance issues have also been investigated: Gibson (2017) argued that Julia has multiple benefits over general-purpose programming languages with respect to graphic rendering capabilities, user experience, and program execution time. Januszek et al. (2018) compared the performance of five programming languages with $\mathcal{O}(n^3)$ algorithms, and observed superior computational efficiency for Julia programs compared to that of Wolfram, R, Python, and C# programs. For parallel programming, Gmys et al. (2020) found Julia programs to outperform C programs with respect to program execution speed.

From the work mentioned above, we observe a lack of research in the domain of defect categorization for Julia programs. We address this research gap in our paper.

3.2.2 Research Related to Defect Categorization

Our paper is also related to prior research that has investigated defect categories for software systems. In 1992, Chillarege et al. (1992a) proposed the Orthogonal Defect Classification (ODC) technique that included eight defect categories. Categories proposed by Chillarege et al. (1992a) were used by Cinque et al. (2014) to categorize defects for air traffic control software. Later in 2008, Seaman et al. (2008a) extended ODC to derive 7 categories of requirements defects, 10 categories of design and source code defects, and 7 categories of test plan defects. Use of existing defect categorization frame-

works, such as ODC and Seaman et al. (2008a)'s work, may be inadequate for Julia, as prior research (2020; 1984) has reported pre-defined defect categorization frameworks to be inappropriate for nascent programming languages and ecosystems. As an example, defect categories mentioned in ODC will not capture Julia-specific defects, such as defects related to type stability as shown in Listing 1.

In contrast, researchers have also constructed bottom-up defect taxonomies for domain-specific software systems. For example, Islam et al. (2019) studied 2,716 SO posts to categorize defects in deep learning libraries, such as Keras and Tensorflow. Humbatova et al. (2020) mined GitHub issues and SO posts to derive a fault taxonomy for software projects that use deep learning. Makhshari and Mesbah (2021) mined 5,565 defect reports to derive a defect taxonomy for the internet of things (IoT) software projects. Rahman et al. (2020) used open coding with commits to derive defect categories for Puppet scripts. Chen et al. (2021) used SO posts to derive a taxonomy of defects for deep learning-based deployment in mobile apps. In short, defect categorization has been an active research area, where researchers have focused on deriving defect categories for domain-specific software systems, such as deep learning, IoT development, and Puppet development. Our work is similar in spirit as we also systematically investigate defect categories for Julia programs.

4 Methodology

In this section, we describe the methodology to conduct our empirical study, which is summarized in Figure 1.

4.1 Repository Mining for Identifying Defects in Julia Programs

We identify defects by mining GitHub repositories. We use GitHub repositories, as it is the most popular social coding website where practitioners host their OSS projects (2018). We posit that by mining repositories from GitHub we will obtain artifacts that describe representative defects in the general Julia ecosystem.

The Julia programming language's design, corresponding compiler [1], and the Julia package manager [2] is hosted on GitHub. As the core components of Julia are hosted on GitHub our hypothesis is that we can find software repositories with sufficient enough of Julia program files by mining open-source GitHub repositories. Empirical data attests to this: we inspected another source code hosting website called GitLab. We observe 60 repositories to contain Julia program files compared to GitHub's 6,474 repositories as shown in Table 1.

[1] https://github.com/JuliaLang/julia
[2] https://github.com/JuliaLang/Pkg.jl

In prior work (Rahman et al., 2018; Agrawal et al., 2018; Krishna et al., 2018; Munaiah et al., 2017), researchers have leveraged a set of attributes to identify repositories that are reflective of professional software development. These attributes include but are not limited to count of certain file types (Rahman et al., 2020; Murphy et al., 2020), count of commits per month (Munaiah et al., 2017), count of issue reports (Rahman et al., 2018; Agrawal et al., 2018), and count of contributors (Agrawal et al., 2018; Krishna et al., 2018). These attributes provide motivation for our criteria to curate Julia repositories:

- **Criterion-1**: At least 10% of the files in the repository must be Julia source code files. The 10% threshold is used as a heuristic to identify repositories that contain sufficient amount of Julia program files. Our assumption is that using this threshold we can identify repositories with sufficient Julia source code files. We do not label a repository exclusively as a 'Python repository' or a 'Julia repository'. Instead, we aim to focus on identifying repositories with sufficient Julia source code files. By using a cutoff of 10% we seek to collect repositories that use Julia.
  This criterion focuses on identifying repositories with sufficient Julia source code files with the '.jl' extension. From our initial exploration of Julia-related repositories we observe repositories that include Julia source code files also include other categories of file types, such as Python files, Makefiles, and JSON files. Our objective is to separate out defects that occur in Julia source code files, and using the threshold of at least 10% we assume that we will identify repositories with decent amount of Julia source code files.
  Let us consider the 'Cxx.jl' repository [3] in this regard. The repository includes Julia source code files along with C and C++ source code files. Our objective is to identify defect categories for Julia source code files but not for C and C++ source code files. According to our criterion we will include this repository as it contains sufficient amount of Julia source code files.
- **Criterion-2**: The repository must be available for download.
- **Criterion-3**: The repository is not a clone to avoid duplicates. Here, cloning refers to forking a repository without modifying repository content.
- **Criterion-4**: The repository must have >= 2 commits per month. Munaiah et al. (2017) previously used the threshold of >= 2 commits per month to determine which repositories have enough software development activity. We use this threshold to filter repositories with little activity.
- **Criterion-5**: The repository has >= 10 contributors. Our assumption is that the criterion of >= 10 contributors may help us to filter out irrelevant repositories, such as repositories used for personal use. Prior research (Humbatova et al., 2020) has also used the threshold of at least 10 contributors.
- **Criterion-6**: The repository has >= 10 *issue reports*. Issue reports are indicative of an active repository where software development happens collaboratively (Agrawal et al., 2018). Using this criterion we assume to filter out repositories used for personal usage.

[3] https://github.com/JuliaInterop/Cxx.jl

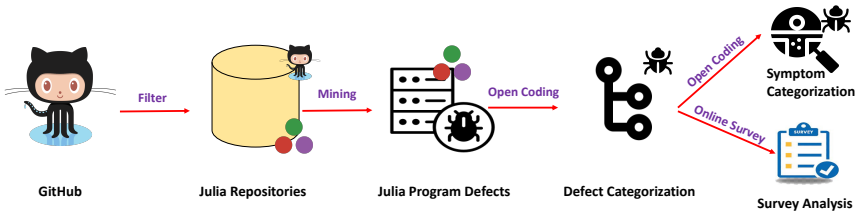

Fig. 1: An overview of our research methodology.

4.1.1 Defect Identification

Using the filtering criteria we identify 112 repositories. A breakdown of the filtering process is provided in Table 1. From the 112 repositories we mine all commits and issue reports to collect defect-related data. Attributes of the 112 repositories is provided in Table 2.

Table 1: OSS Repositories Satisfying Criteria

Initial Repo. Count	3,405,303
Criteria-1 (10% Julia files)	6,474
Criteria-2 (Available)	6,474
Criteria-3 (Not a clone)	3,872
Criteria-4 (Commits/Month $\geq$ 2)	1,173
Criteria-5 (Contributors $\geq$ 10)	253
Criteria-6 (Issue reports $\geq$ 10)	112
Final Repo Count	112

Table 2: Attributes of Repositories

Attribute	**Value**
Count of repositories	112
Count of Julia files	3,668
Count of commits	280,287
Count of Julia-related commits	30,494
Count of contributors	2,566
Count of issue reports	3,038
Count of stars	24,391
Duration	01/2014-01/2022
Julia-related size (Total SLOC)	1,156,608

First, we apply a keyword search to identify commit messages and issue reports that reports a defect in a Julia program. We apply the keyword search for 30,494 commit messages, and the titles, descriptions, and comments for each of the 3,038 issue reports. We use the following keywords, which also have been used in prior defect categorization research (Garcia et al., 2020;

Rahman et al., 2020): 'bug', 'defect', 'error', 'fault', 'fix', 'flaw', 'incorrect', 'issue', and 'mistake'. Using our keyword search we identify 7,689 commits and 1,492 issue reports.

Second, we apply qualitative analysis to identify defects from the collected 7,689 commits and 1,492 issue reports. Our keyword-based approach can generate false positives, which necessitates application of qualitative analysis. We use 2 types of artifacts to identify defects: (i) messages and corresponding diffs from the set of 7,689 commits, and (ii) entire content of each of the 1,492 issue reports. For each of these 2 types of artifacts, 2 raters individually identify if one or multiple Julia-related defects appear in the artifact. To identify defects, both raters use the IEEE definition: "*an imperfection or deficiency in the code that needs to be repaired*". For defect identification, the raters inspected (i) if problematic code exists in the artifact, (ii) if problematic code leads to an immediate incorrect or undesired consequence upon execution that is explicitly expressed by a developer, and (iii) if the problematic code was repaired.

The first and second authors acted as raters. The first and second authors respectively have 10 and 2 years of experience in software development. Upon completion of the inspection process, we calculate Krippendorff's α (2018) to quantify agreement, similar to prior work in software engineering (Antinyan et al., 2017; Raulamo-Jurvanen et al., 2019; Friess, 2019; Catolino et al., 2019). The Krippendorff's α is 0.53, indicating 'unacceptable' agreement (Krippendorff, 2018). Both raters discussed their disagreements and identified the cause of disagreements to the perception of features or defects. Upon discussion, both raters conduct the inspection process again. After completing the inspection process, we calculate Krippendorff's α to be 1.0, indicating 'perfect' agreement (Krippendorff, 2018). We use Krippendorff's α instead of Cohen's κ, because Krippendorff's α: (i) emphasizes disagreement leading to more reliability on the achieved agreement rate, and (ii) handles multiple categories (Krippendorff, 2018). Furthermore, qualitative analysis experts have advocated for the use of Krippendorff's α over Cohen's κ (Krippendorff and Fleiss, 1978; Lombard et al., 2010).

In all, we identify 742 defects from 740 commits and 52 issue reports. Out of 740 commits, 52 commits had a mapping to 52 issue reports, from which we identify 52 defects. Rest of the 690 defects were obtained from 690 commits. On average we identify 6.6 defects per repository (min, median, max = 1.0, 5.0, 98).

4.2 RQ1: Defect Categorization

We answer **"RQ1: What categories of defects exist in Julia programs?"** by applying qualitative analysis to categorize the 742 defects identified from Section 4.1.1. We use open coding, a qualitative analysis technique that summarizes the underlying theme from unstructured text (Saldaña, 2015). We apply a multi-phase open coding process with two phases following prior work on fault categorization (Chen et al., 2021). According to researchers, multi-phase

```
1 -        if regime_switching
2 +        if haskey(get_settings(m), :time_varying_trends) && get_setting(m,
  ↪  :time_varying_trends)
3 +            start_date = get(get_setting(m, :shockdec_startdate))
4 +            end_date = max(prev_quarter(date_forecast_start(m)),
  ↪  date_forecast_end(m))
5 +            return (ndraws, nvars, DSGE.subtract_quarters(end_date,
  ↪  start_date)+1)
6 +        elseif regime_switching
```

Listing 3: An example of a commit message for which raters disagreed during synchronized open coding.

coding is pivotal to gain multiple perspectives, ensure rater reliability, and achieve rater consensus for qualitative analysis (Sweeney et al., 2013; Hickey and Kipping, 1996).

4.2.1 Synchronized Open Coding

In the case of synchronized open coding, two raters identify defect categories together. The two raters are the first and second authors of the paper. They apply open coding on randomly selected 370 commits and 26 issue reports that correspond to 371 defects identified from Section 4.1.1. Both raters read each artifact in its entirety to understand the context of defects and assign each artifact with initial codes.

Next, both raters group similar codes into categories. The grouping process is iterative where both raters went back and forth to reach an agreement. Upon completion, both raters reached an agreement with respect to defect categorization on all but 10 defects, for which the third author acted as the resolver. The third author is a professional software engineer with 3 years of experience in Julia. The resolver's decision is final for the 10 disagreements. Krippendorph's α was 0.84, indicating an 'acceptable' agreement.

Disagreement example: As shown in Listing 3, for the commit message `fix dimension mismatch in trends forecasting` the raters disagreed, as one rater identified this commit message to be related with array, whereas the other rater labeled the defect as a conditional defect. The defect was resolved as a conditional defect as the code changes displayed changes in if-else, i.e., conditional statements.

4.2.2 Independent Open Coding

In this phase, the two raters independently apply open coding. Similar to the synchronized open coding phase, the two raters are the first and second authors of the paper. They apply open coding on the remaining 370 commits and 26 issue reports that correspond to 371 defects identified from Section 4.1.1. Similar to the synchronized open coding phase, for each artifact, both raters

```
1 - ccall(:memcpy, Ptr{Cvoid}, (Ptr{Cvoid}, Ptr{Cvoid}, UInt), pr_p, v,
  ↪  nnz*sizeof(V))
2 + copyto!(unsafe_wrap(Array, pr_p, (nnz,)), v)
```

Listing 4: An example of a commit message for which raters disagreed during independent open coding.

read all content to understand the context of defects and assign each artifact with initial codes to identify the category of the defect.

Upon completion of this phase, we record a Krippendorff's α of 0.87, indicating an 'acceptable' agreement. The raters disagree on the defect categories for 11 defects that are resolved using the resolver, i.e., the third author of the paper. The resolver's decision is final on the disagreed upon defects.

Disagreement example: As shown in Listing 4, for the commit message `fix segfault due to memcpy (#155) fixes failing tests on linux and osx` the raters disagreed, as one rater identified this commit message to be polyglot-related, whereas the other rater labeled the defect as an array defect. The defect was resolved as a polyglot defect as the code changes showcased the characteristics of a polyglot defect.

4.3 RQ2: Defect Symptoms

We answer **"RQ2: What are the symptoms of defects in Julia programs?"** by identifying defect symptoms for each of the defect categories in two steps: *first*, for each category we separate defects that map to that category. *Second*, following prior work (Zhang et al., 2018a; Di Franco et al., 2017) we identify incorrect or undesired results due to the defect expressed in the commit message or the issue report, as a symptom. *Third*, from the associated artifacts for each defect in the category we apply open coding in two phases. In the first phase, the two raters apply synchronized open coding, whereas in the second phase the raters apply independent open coding. In the case of synchronized open coding, we use 50% of the defects that belong to each defect category identified from Section 4.2. The rest of the 50% defects are used for independent open coding. While conducting open coding both rater identify and categorize defect symptoms, i.e., consequences of defects as described in the artifact of interest. We repeat the process for all defect categories.

Upon completion of this phase, we record a Krippendorff's α of 0.92, indicating an 'acceptable' agreement. The raters disagree on the symptom categories for 7 defects that are resolved using the resolver who is the third author of the paper. The resolver's decision is final on all disagreements.

Disagreement example: For the commit message `fix segfault during finalization (#615) * fix typo` the raters disagreed on the symptom, as one rater reported to consequence for this commit messages, whereas the other rater reported 'crash' as a consequence. The symptom was resolved as a crash as segfaults or segmentation faults lead to a crash.

4.4 RQ3: Perceptions of Defect Categories

We answer **"RQ3: How do developers perceive the identified defect categories for Julia programs?"** by conducting an online survey with developers who write Julia programs. In the survey, we first ask developers about their experience in writing Julia programs. Next, we describe each of the identified defect categories with definitions. We then ask questions related to perceived frequency and severity: *first*, we ask "*How frequently do the identified defect categories occur in Julia programs?*". Survey participants used a five-item Likert scale to answer this question: 'Not at all frequent', 'Rarely', 'Somewhat frequently', 'Frequently', and 'Highly frequent'. *Second*, we ask *"What is the severity of the identified defect categories"* To answer this question, survey participants used the following five-item Likert scale: 'Not at all severe', 'Low severity', 'Moderately severe', 'Severe', and 'Highly severe'. We use a five-item Likert scale for both questions following Kitchenham and Pfleeger's guidelines (2008). Furthermore, following Kitchenham and Pfleeger (2008)'s advice we apply the following actions before deploying the survey: (i) add explanations related to the purpose of the study, (ii) add instructions on how to complete the survey, (iii) add explanations related to preservation of confidentiality, (iv) provide an estimate of completion time, and (v) conduct a pilot survey to get initial feedback. From the feedback of the pilot survey, we add an open-ended text box so that survey respondents can provide more context for their responses. The survey questionnaire is included in our verifiability package (Rahman, 2022).

We select our participants by collecting email addresses of practitioners who have developed Julia programs in our collection of 112 OSS repositories. In all, we deploy our survey to randomly-selected 200 practitioners via e-mails. We offer a drawing of one 50 USD Amazon gift card [4] as an incentive for participation following Smith et al. (2013)'s recommendations. We conduct the survey from November 2021 to March 2022 following the Internal Review Board (IRB) protocol #2234.

According to our IRB approval process, we made sure that we did:

- Not release any private and sensitive information of the survey participants;
- Seek approval from each participant via prior to sending the survey;
- Not commercially advertise any existing Julia-related research of the research group;
- Not use automation to send emails. Instead, we send personalized email messages where we made it clear that the purpose of the email is only to seek feedback on our research;
- Provide full identify of the lead researcher who is conducting the survey; and
- Explicitly mention that participation or lack thereof will not impact their occupation.

[4] https://www.amazon.com/gift-cards

As part of setting up and configuring a GitHub repository if a GitHub user wants to keep their email address private they will select the "Keep my email addresses private" feature (github, 2023), which prevent exposure of the developer's email in the Git logs. The availability of the developers' emails in the Git logs is an indication of a developer not explicitly opting for the "Keep my email addresses private" feature.

5 Answer to RQ1: Defect Categorization

As summarized in Figure 2, we identify 9 defect categories that we describe next alphabetically:

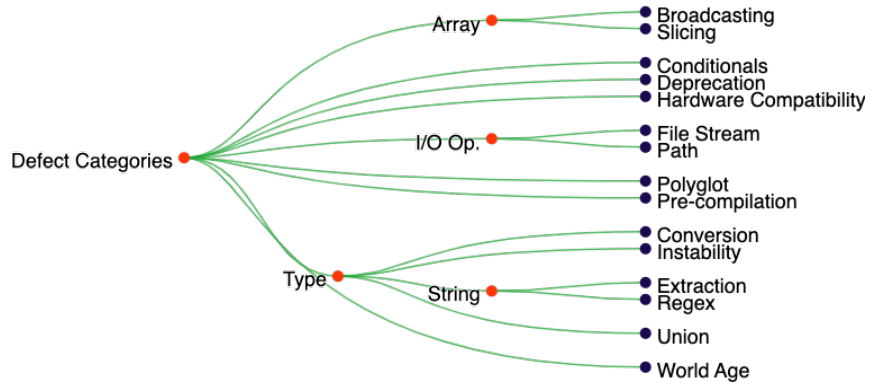

Fig. 2: Defect categories for Julia programs.

I. Array: Defects that occur due to incorrect usage of arrays in Julia programs. We further identify two sub-categories:

I*a. Broadcasting:* Defects that occur when using broadcasting with arrays. In Julia, broadcasting is the feature of performing element-by-element operations on arrays, e.g., adding a vector to each column of matrix (Jul, 2022d). Broadcasting enables developers to perform such operations without replicating the vector to the size of the matrix.

Example: As shown in Listing 5, a developer incorrectly applied array broadcasting by missing the '.' operator (tknopp, 2018). Because of this defect the implementation of sampling density was incorrect. Sampling density is a metric that measures the count of recorded samples per unit distance when an analog signal is being converted to a digital signal (Merchant and Castleman, 2005).

I*b. Slicing:* Defects that occur when a sub-array is extracted from an array in a Julia program. Defects related to array slicing are manifested in form of unwanted memory allocations.

Example: A Julia program used to simulate a Markov Chain was erroneous (amckay, 2016) included a defect related to array slicing. A Markov

```
for i in 1:iters
 - p.tmpVec[:] = 0.0
 + p.tmpVec[:] .= 0.0
  convolve_adjoint!(p, weights, p.tmpVec)
  ...
```

Listing 5: An example of an array broadcasting defect in a Julia program.

Chain is a stochastic model that describes a a sequence of possible events where the probability of each event depends on the state recorded in the previous event (Gagniuc, 2017). The defect was repaired by accurately extracting the sub-array, as shown in Listing 6.

```
for i in 1:k
    for t in 1:ts_length-1
-       X[t+1, i] = draw(P_dist[X[t]])
+       X[t+1, i] = draw(P_dist[X[t,i]])
    end
end
```

Listing 6: An example of an array slicing defect in a Julia program.

II. Conditionals: Defects that appear because of incorrect conditional logic in a Julia program. Incorrect conditional logic either corresponds to providing incorrect values, incorrect operators, or a combination of both.

Example: Listing 7 shows an example of a conditional defect, which results in incorrect calculation of reservoir sampling (Drvi, 2020). Reservoir sampling is a collection of randomized algorithms for choosing a simple random sample without replacement (Vitter, 1985).


```
  j = rand(1:o.n)
- if j < length(o.value)
+ if j <= length(o.value)
    o.value[j] = y
  end
```

Listing 7: An example of a conditional defect.

III. Deprecation: Defects that occur for using Julia language constructs that have been deprecated, and needs to be avoided. Since its inception in

2012, the Julia programming language has undergone a series of changes. As a result of these changes, code constructs that were previously compatible with the Julia compiler, no longer work. As of February 2022, Julia is using version 1.7, and for any code construct that is older than 1.6, the Julia compiler will generate a warning message. For example, `Uint64` was allowed in Julia 0.3, but in Julia 1.7, `UInt64` must be used to avoid a deprecation warning (Julia, 2021). According to the Julia documentation "*a deprecated function internally performs a lookup in order to print a relevant warning only once. This extra lookup can cause a significant slowdown, so all uses of deprecated functions should be modified as suggested by the warnings*" (Jul, 2022d).

Example: In Listing 8 we observe a developer to use the outdated syntax for `Symbol`, which resulted in a compiler warning (yuyichao, 2016). In Julia, `Symbol` is used for meta-programming, which allows the Julia compiler to represent its own code as a data structure of the language itself. In Listing 9, we provide another example that we have categorized as a deprecation defect. We observe a practitioner to modify a code snippet `Base.REPL.run_interface()`, which is outdated, and later replaced with `REPL.run_interface()`.

```
for T in (:Date, :DateTime, :Delta)
...
-   f = symbol(string("Py", T, "_Check"))
+   f = Symbol(string("Py", T, "_Check"))
    @eval $f(o::PyObject) = pyisinstance(o, $t)
end
```

Listing 8: An example of a deprecation defect.


```
- Base.REPL.run_interface(Base.active_repl.t,
↪   Base.LineEdit.ModalInterface([panel]))
+ REPL.run_interface(Base.active_repl.t,
↪   Base.LineEdit.ModalInterface([panel]))
```

Listing 9: Example of another deprecation defect.

IV. Hardware Compatibility: Defects that occur due to hardware-related compatibility issues. Not all hardware devices are compatible with Julia, which can lead to program failures.

Example: In an issue report (JuliaLang/IJulia.jl, 2017), we document an example of a hardware compatibility defect. A developer describes how a Julia program crashed when trying to run the program on a Raspberry Pi device.

The defect occurred due to (i) not allocating a swap space, and (ii) not allocating a RAM of 1.3GB on the Raspberry Pi device. While Julia provides support for x86 and ARM processors, we observe developers to face challenges when executing Julia programs on Raspberry Pi devices.

V. I/O Operation: Defects that occur when working with input/output (I/O) streams and objects. We identify two sub-categories:

V*a. Streams:* Defects related to I/O streams in Julia programs that occur due to incorrect application of stream-related code constructs.

Example: We document a stream-related defect for 'MentaLiST', a Julia-based project developed to analyze pathogen outbreak (WGS-TB, 2021b). As shown in Listing 10, not closing a stream using `close()` resulted in a crash (WGS-TB, 2021a).

```
function read_alleles(fastafile, ids)
    ...
-
+   close(fh)
  return alleles
end
```

Listing 10: An example of a stream defect.

V*b. Path:* Defects related to specification of paths for files and directories upon which the execution of a Julia program is dependent.

Example: Incorrect specification of a file path created a crash (JuliaLang, 2017). Listing 11 shows the defect, which we document for 'IJulia', a notebook interpreter for developing Julia programs (JuliaLang, 2021).


```
for problem in problems
...
-   include(joinpath(nlpmodels_path, "$problem_s.jl"))
+   include(joinpath(nlpmodels_path, "problems", "$problem_s.jl"))
nlp = CUTEstModel(uppercase(problem_s))
adnlp = eval(Meta.parse("$(problem)_autodiff"))()
```

Listing 11: An example of a stream defect.

VI. Polyglot: Defects that occur when developers incorrectly use Julia-provided utilities to interface with programs written in non-Julia languages, such as C and Fortran. Julia includes utilities, such as `@ccall` to interface with functions

written in C and Fortran (Jul, 2022d). The Julia documentation provides instructions on how to correctly use these utilities, which developers do not abide by and introduce defects in their programs.

Example: A polyglot defect is shown in Listing 12. According to the issue report (JuliaSmoothOptimizers/CUTEst.jl, 2015), the developer directly called `@ccall`, which resulted in a crash. The correct approach is to indirectly call `@ccall` with `@dlsym` (abelsiqueira, 2015).

```
function bar!(nlp, goth, x, v)
  ...
  - @eval ccall(("cutest_uhprod_", $(nlp.libname)), Void,
  + ccall(@dlsym(:cutest_cfn_,mylib), Void, ...
    return nlp.Hv
end
```

Listing 12: An example of a polyglot defect.

VII. Pre-compilation: Defects that occur due to incorrect usage of Julia's pre-compilation feature. Julia provides the option of using `__precompile()__` so that external Julia packages specified as dependencies can be loaded only once and stored in a cache. When the Julia program is executed again, the Julia compiler will fetch contents from the cache, and use the cached content to execute the program. In this manner, Julia does not need to compile the external Julia dependencies every time the program is being executed.

Example: We observe a pre-compilation defect in an OSS repository to generate Vega-based visualizations. Vega is used to implement grammars for graphics, similar to that of ggplot2 (Satyanarayan et al., 2016). As shown in Listing 13, by enabling pre-compilation with `__precompile__` the developer reported four times performance improvement (randyzwitch, 2015).


```
-
+VERSION >= v"0.4-" && __precompile__()
module Vega
  using JSON, ColorBrewer, KernelDensity, NoveltyColors, StatsBase,
  ↪ Parameters, Missings
 import Base: print, show
```

Listing 13: An example of a pre-compilation defect.

VIII. Type: Defects that occur due to incorrect use of types. We identify the following sub-categories:

VIII*a. Conversion:* Defects that occur when a developer either attempts to convert one type to another, or uses an incorrect type, which necessitates conversion to the correct type.

Example: As documented in an issue report (JuliaSmoothOptimizers, 2019), a crash occurred because of multiplying a `Float64` matrix to a `ComplexFloat64` array. As shown in Listing 14 the defect was fixed by using type promotion, the feature of converting values of mixed types to a single common type (Jul, 2022d).

VIII*b. Instability:* Defects that occur when a Julia program is not type stable. Type stability refers to the property of a Julia program where the type of every variable does not vary at runtime (Jul, 2022d). The Julia compiler has to support any of the 221 subtypes for variables whose types vary during runtime. This results in unnecessary memory allocation. Writing type stable code is considered as a "*key to performant Julia code, allowing the compiler to optimize*" (Bezanson et al., 2018). Type stability allows the Julia compiler to determine the type of a variable at compile, and is strongly recommended for rapid execution of Julia programs.

Example: As shown in Listing 15, type instability occurred due to not specifying the types of `n` explicitly. This allows for the variable `n` to have varying types during runtime, leading to type instability. The defect was repaired by explicitly providing the type for `n` with `::Int` (simonster, 2014).

VIII*c. String:* Defects that occur when using String data types in Julia programs. We identify two sub-categories: (i) character extraction defects: defects that occur while extracting one or a sequence of characters from a string,


```
- X = reshape(convert(Vector{T}, x), p, m)
+ S = promote_type(T, eltype(x))
+ X = reshape(convert(Vector{S}, x), p, m)
  return Matrix(B * X * transpose(A))[:]
end
function tprod(x)
```

Listing 14: An example of a type conversion defect.

```
function read_array(obj::JldDataset, dtype::HDF5Datatype, T::Type,
  ↪  dspace_id::HDF5.Hid)
...
- n = prod(dims)
+ n = prod(dims)::Int
  h5sz = sizeof(dtype)
  out = Array(T, dims)
```

Listing 15: An example of a type instability defect.

```
@render i::Inline x::AbstractString begin
- Row(span(".syntax--string", c("\""), render(i,
↪   Text(escape_string(x[1:chr2ind(x, 500)])))))),
+ Row(span(".syntax--string", c("\""), render(i,
↪   Text(escape_string(join(x[1:min(length(x),500)])))))),
  Text("..."))
end
```

Listing 16: An example of a character extraction defect.

and (ii) regex defects: defects that occur when using regular expressions with strings.

Example: Character extraction: As shown in Listing 16, a character extraction defect occurred when a developer used `chr2ind` to iterate over the characters by obtaining their `byte` indices instead of character indices. The defect, which resulted in incorrect rendering was repaired by obtaining character indices with `length(x)` (MikeInnes, 2017).

Regular expression (Regex): We document a regex defect in a commit (MOSEK, 2018) shown in Listing 17, which provided the wrong pattern that needs to be matched using regular expressions. The defect resulted in a build error.

```
txt = readstring(`$bindir/$mosekbin`)
- m = match(r"\s*MOSEK Version ([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)",txt)
+ m = match(r"\s*MOSEK Version ([0-9]+\.[0-9]+\.[0-9])",txt)
if txt == nothing
    return nothing
else
```


Listing 17: An example of a regex defect.

VIII*d. Union:* Defects that occur due to inadequate use of type unions. With the `Union` keyword, Julia allows developers to create a custom abstract type that includes objects as specified by its argument types. `Unions` could be useful to declare `nullable` types, which allows to set a special value `NULL`, for example as done in the Structured Query Language (SQL) (Mauny and Vaugon, 2014).

Example: We document a union defect for an OSS repository used to implement data structures, such as red-black trees. In Listing 18 we document a defect where the developer incorrectly declared a `nullable` type, which was repaired with `Union(K, Nothing)` (eulerkochy, 2020).

IX. World Age: Defects that occur when the `world age` principle of Julia is violated. In Julia, `world age` is defined as a mechanism that determines

```
mutable struct RBTreeNode{K}
  color::Bool
- data::K
+ data::Union{K, Nothing}
  ...
  RBTreeNode{K}(d::K) where K = new{K}(true, d, nothing, nothing, nothing)
end
```

Listing 18: An example of a `Union` defect.

the set of method definitions that are reachable from the currently executing method (Belyakova et al., 2020). With `world age`, the Julia compiler assigns each method definition an age, and for each function call, the compiler checks if the `world age` for the program is greater than the age of the method that is about to be invoked (Belyakova et al., 2020). If the age of a method definition is greater than the `world age`, the program fails. `World age` helps the Julia compiler to facilitate dynamic code loading as well as perform program optimizations (Belyakova et al., 2020).

Example: In an OSS repository, we document evidence of a `world age` defect, which manifested in a program failure (pfitzseb, 2017). As shown in Listing 19, for the `Atom.connect` method, the Julia compiler detected a violation of `world age` stating "*The applicable method may be too new: running in world age 21578, while current world is 21595*". For the `Atom.connect()` method the age is 21595, which is higher than that of the `world age (21578)`. The defect was repaired by using `eval()`, which decreased the age of `Atom.connect` (pfitzseb, 2017).


```
function connect(args...; kws...)
  activate()
- Atom.connect(args...; kws...)
+ eval(:(Atom.connect($args...; $kws...)))
  return
end
```

Listing 19: An example of a world age defect.

Frequency: We report the frequency of each defect category in Table 3. Each cell presents the defect count for each category. 'Total' represents the total for each category. Type is the most frequent defect category.

For types, conversion defects are dominant: of the identified type-related defects, 209, 21, 11, and 1 are respectively, conversion, instable, string, and `Union` defects. Of the 11 string defects, 2 are regex and 2 are extraction defects.

Table 3: Frequency of Defect Categories in Julia Programs

Category	Defect Count
Array	114
Conditionals	103
Deprecation	210
Hardware Compatibility	1
I/O Operation	45
Polyglot	21
Pre-compilation	5
Type	242
World Age	1
Total	742

Of the identified 114 array defects, 8 and 106 are respectively, broadcasting and slicing defects. Of the identified 45 I/O operation defects, 3 and 42 are respectively, file stream and path defects.

We also compare the defect categories identified for Julia programs with existing taxonomies. We review prior work on defect categorization and identified if one or multiple defect categories identified from our qualitative analysis also appear for other software systems. We select a set of papers that include defect taxonomies for non-Julia 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:

- "A Comprehensive Study of Autonomous Vehicle Bugs" (Garcia et al., 2020)
- "An Empirical Study on TensorFlow Program Bugs" (Zhang et al., 2018b)
- "Bug characteristics in open source software" (Tan et al., 2014)
- "Defect Categorization: Making Use of a Decade of Widely Varying Historical Data" (Seaman et al., 2008b)
- "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts" (Rahman et al., 2020)
- "IoT Bugs and Development Challenges" (Makhshari and Mesbah, 2021)
- "Not All Bugs Are The Same: Understanding, Characterizing, and Classifying Bug Types" (Catolino et al., 2019)
- "Orthogonal Defect Classification: A Concept for In-process Measurements " (Chillarege et al., 1992b)
- "Taxonomy of Real Faults in Deep Learning Systems" (Humbatova et al., 2020)

The papers related to defect categorization can be divided into two groups:

- Generic software systems: the defect taxonomies presented in the following papers, "Orthogonal Defect Classification: A Concept for In-process Measurements" (Chillarege et al., 1992b), "Not All Bugs Are The Same: Understanding, Characterizing, and Classifying Bug Types" (Catolino et al., 2019), "Bug characteristics in open source software" (Tan et al., 2014), and "Defect Categorization: Making Use of a Decade of Widely Varying

Historical Data" (Seaman et al., 2008b) are applicable for generic software projects. Amongst these three publications, the two papers namely, "Orthogonal Defect Classification: A Concept for In-process Measurements" (Chillarege et al., 1992b), "Bug characteristics in open source software" (Tan et al., 2014), and "Defect Categorization: Making Use of a Decade of Widely Varying Historical Data" (Seaman et al., 2008b) are seminal publications with high impact in the domain of software engineering research. Our hypothesis is that if Julia-related defect categories are generic then identified defect categories will overlap with the defect categories reported in these publications.
- Specialized software systems: the defect taxonomies presented in the following papers "IoT Bugs and Development Challenges", "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts", "A Comprehensive Study of Autonomous Vehicle Bugs", "Taxonomy of Real Faults in Deep Learning Systems", and "An Empirical Study on TensorFlow Program Bugs" respectively, present defect categories for IoT, infrastructure as code, autonomous vehicles, 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 software systems, there might be some Julia-related defect categories that overlap with one or multiple of these five software systems.

By considering representative publications from these two groups we aim to gain a foundational understanding of the existing defect categories, which further enable us to compare our derived Julia-related defect categories to that with existing defect categories for previously studied software systems.

We report the defect categories of other non-Julia software systems in Table 4. We observe three defect categories that have not been reported in prior software systems: polyglot, pre-compilation, and world age.

Sanity Check: Our defect-related keyword search is susceptible to conclusion validity as it might miss commit messages for which unreported defect categories exist but are not identified. We mitigate this limitation by determining if new defect categories appear for commit messages that are not identified through our keyword search process. We use a sample with 95% confidence interval, 5% margin of error, and 50% population proportion from the set of 22,806 commits. We end up with a sample of 378 commits. For this set of commits the first author apply independent open coding to determine if any defect categories are identified that are not reported in Section 5. For performing open coding the first author inspected the commit message, corresponding code diff, and any referenced issue.

From the set of 378 commits we identify four defect-related commits, of which three can be labeled as type-related defects, and one can be labeled as a conditional defect. We list these commit messages below with appropriate references.

Table 4: Appearance of Defect Categories in Previously-studied Software Systems

Category	Previously-studied Software System
Array	Deep Learning Projects (Humbatova et al., 2020; Zhang et al., 2018b), Linux Kernel (Tan et al., 2014)
Conditionals	IBM Proprietary Software (Chillarege et al., 1992b), NASA Software Projects (Seaman et al., 2008b), Autonomous Vehicle (Garcia et al., 2020), Puppet Manifests (Rahman et al., 2020), Linux Kernel (Tan et al., 2014)
Deprecation	Deep Learning Projects (Humbatova et al., 2020; Zhang et al., 2018b), Eclipse Projects (Catolino et al., 2019)
Hardware Compatibility	IoT Software (Makhshari and Mesbah, 2021)
I/O Operations	IBM Proprietary Software (Chillarege et al., 1992b), NASA Software Projects (Seaman et al., 2008b), Linux Kernel (Tan et al., 2014)
Polyglot	Not reported for prior software systems
Pre-compilation	Not reported for prior software systems
Type	Not reported for prior software systems
World age	Not reported for prior software systems

– `update to latest type inference`: this commit [5] is a conditional defect.
– `explicitly convert from cuint to cint`: this commit [6] is related to type conversion.
– `Correct return type of hwloc_topology_destroy`: this commit [7] is related to type conversion.
– `get rid of type instabilities`: this commit [8] is a type instability defect.

6 Answer to RQ2: Defect Symptoms

We identify 7 symptoms, which we present next alphabetically:
I. Build Failure: This symptom refers to failures during a build, i.e., the process of converting Julia programs into executable software artifact.
II. Crash: This symptom refers to a Julia program's execution being terminated abruptly.
III. Incorrect Calculation: This symptom occurs when a Julia program executes but provides incorrect calculations. Examples of consequences include but are not limited to (i) consequences of defects that result in incorrect statistical models, such as fixed effects, generalized linear models (GLMs), and interaction effects; (ii) consequences of defects that result in incorrect

[5] https://github.com/JunoLab/Atom.jl/commit/4650266b3c2a74d2f6db61e8710ae5b2cd8593a9
[6] https://github.com/JuliaParallel/Hwloc.jl/commit/cdcf861c5ab5fa9de876344c7e2271ec69e9b916
[7] https://github.com/JuliaParallel/Hwloc.jl/commit/65781b7c08ee0349ccf7db6c629730beb87e8625
[8] https://github.com/JunoLab/Atom.jl/commit/080609b34d4c06dfb13950f9cdd375495d25c7f5

Table 5: Frequency of Defect Symptoms in Julia Programs

Category	Build Failure	Crash	Incorrect Calculation	Incorrect Render	Speed Reduction	Test Failure	Warning
Array	0	1	111	0	1	1	0
Conditionals	1	0	69	15	2	16	0
Deprecation	1	2	0	0	2	4	201
Hardware	0	1	0	0	0	0	0
I/O Operation	9	17	2	2	0	15	0
Polyglot	0	13	3	4	0	0	1
Pre-compilation	0	3	0	0	2	0	0
Type	6	103	26	33	36	37	1
World Age	0	0	0	0	1	0	0
Total (Symptom)	17	140	211	54	44	73	203

matrix-related computations; and (iii) consequences of defects that result in an incorrect simulation of circuit elements, such as resistors and diodes.

IV. Incorrect Rendering: This symptom occurs when a Julia program renders strings or colors in user interfaces, which does not meet end-user expectations.

V. Speed Reduction: This symptom occurs when a program's execution speed is reduced due to a defect.

VI. Test Failure: This symptom refers to test failures due to defects in Julia programs.

VII. Warning: This symptom refers to compiler-generated warning messages for Julia programs.

Symptom Frequency: We quantify the frequency of the identified symptoms in Table 5. Each cell represents the count of defects for which the consequence appears. For example, according to Table 5 the consequence of one array-related defect is a crash. We observe incorrect calculations to be the most frequently occurring symptom.

We acknowledge that the identified defect symptoms are applicable for other software systems. However, by identifying these defect symptoms we can understand what categories of defects is related with symptom categories. Such understanding can help us contextualize the consequences of the identified defect categories both generic, as well as unique to Julia programs.

7 Answer to RQ3: Perceptions of Identified Defect Categories

We answer RQ3 by providing results related to developer perceptions on the frequency and severity of the identified defect categories. We obtain 52 responses in total. The median reported experience in Julia is 4.5 years. In Figures 3 and 4 we respectively, report developer perceptions for frequency and severity of the identified defect categories. The x and y-axis respectively

presents the percentage of survey participants and defect categories. For example, from Figure 3 we observe 48% of the total survey respondents to identify array manipulation as a 'frequently' or 'highly frequent' defect category. From Figures 3 and 4, we observe developers to perceive type-related defects to be the most frequent and the most severe defect category.

In response to our question related to unidentified defect categories, one survey respondent mentioned 'wrong dispatch'. Dispatching refers to Julia's multiple dispatch feature: Julia allows functions to have multiple definitions as long as each definition uses a combination of arguments that are different from one function to another (Jul, 2022d). According to the survey respondent, developers use Julia's multiple dispatch feature incorrectly, which leads to defects. One respondent mentioned defects in LLVM (Lattner and Adve, 2004), which is leveraged by the Julia compiler to generate intermediate representations. The survey respondent stated: "*Julia uses LLVM as an intermediate representation, and LLVM itself has a few defects*".

From the open-ended textbox, we also observe survey respondents provide reasons on why they agree with the identified defect categories. One respondent agreed with all identified defect categories stating all identified categories to "*seem reasonable*". The respondent was referring to the fact that they agree that all identified defect categories are in fact Julia-related, but the frequency can vary. One respondent agreed with polyglot defects stating referring to C code from Julia with callbacks is challenging: "*Tons of issues trying to make a C library binding in Julia which uses callback functions that take user pointers. Specifically, I'm getting segfaults or undefined behavior when the callback is called. The library we are writing the wrapper for, also supports calling callbacks in separate threads, but that seems nearly impossible to do safely in Julia*". Another respondent reflected on the experience with hardware:"*Most Julia defects I've encountered stem from odd setups. Hardware defects that only show themselves in the presence of some instructions emitted by Julia, e.g., AVX instructions on specific CPU/motherboard combinations.*". In the case of pre-compilation defects, one respondent mentioned how it can cause delays when deployed to cloud-based instances "*I've also experienced a lot of difficulty trying to build executable packages or Julia system images from our code to deploy on cloud instances to remove startup time*".

In Table 6 we report how many of the practitioners perceive the combination of defect categories to be 'Frequent' or 'Highly Frequent'. For space constraints we report >= 5% of the practitioners who found combination of categories to be frequent. The full table is available in our replication package (Rahman, 2022). From Table 6 we observe 42.3% of the survey respondents to agree that both array and conditional defects frequently occur. We also observe 38.4% of the survey respondents to perceive array, conditional, and type-related defects to be frequent. We observe no practitioner to find all defect categories to be equally frequent.

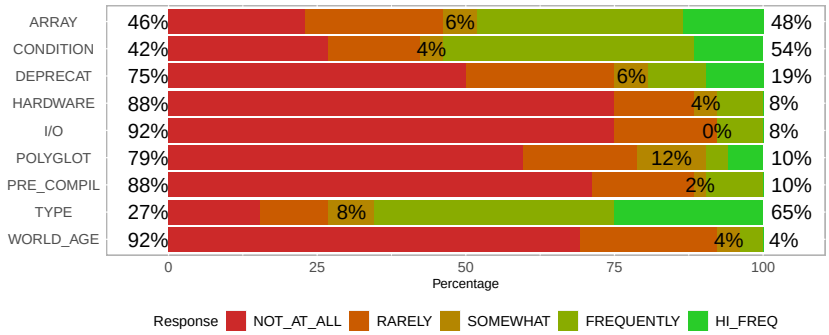

Fig. 3: Perceived frequency of identified defect categories.

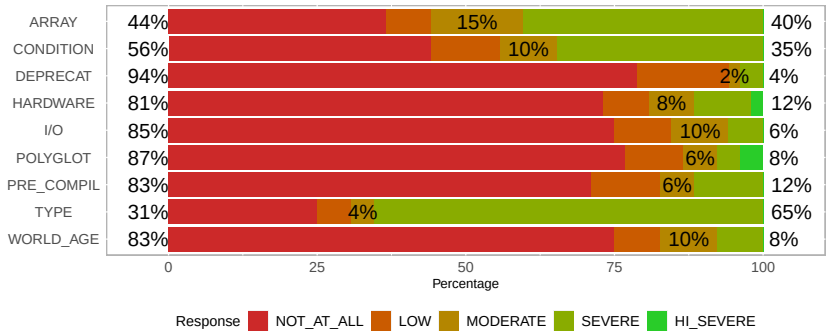

Fig. 4: Perceived severity of identified defect categories.

8 Discussion

We discuss the findings of our empirical study as follows:

Symptom-related Implications: Symptom categories reported in Section 8 demonstrates the importance of identified defect categories. For example, we observe crashes to be commonplace for Julia-specific categories, as well as for categories applicable for other languages. The existence of incorrect-calculation symptoms shows unmitigated array and conditional defects have serious consequences for scientific computing and statistical modeling—areas that the Julia community engages with. Also, problems related to program execution speed show developers to not obtain the desired benefits from Julia-provided utilities.

***Came for Syntax and Speed, Now Need Help*——Implications for Toolsmiths**: Our results presented in Section 5 provide directions on how to improve tooling for Julia programs:

Automated detection of Julia anti-patterns: Our findings show that developers violate recommended coding practices provided by the Julia documentation. Not adopting these recommended practices can introduce defects, such

Table 6: Practitioner-reported Frequency for Defect Category Combinations

Category Combination	Practitioner (%)
Array, Conditional	42.3%
Array, Type	40.4%
Conditional, Type	40.4%
Array, Conditional, Type	38.4%
Conditional, Deprecation	13.5%
Deprecation, Pre-compilation	13.4%
Array, Deprecation	11.5%
Array, Conditional, Deprecation	11.5%
Conditional, Deprecation, Type	11.5%
Array, Deprecation, Type	11.5%
Array, Conditional, Deprecation, Types	11.5%
Polyglot, Type	9.6%
Array, Hardware	7.7%
Conditional, Hardware	7.7%
Hardware, Type	7.7%
I/O Operation, Type	7.7%
Conditional, Hardware, Type	7.6%
Array, Hardware, Type	7.6%
Array, Conditional, Hardware	7.6%
Array, Conditional, Hardware, Type	7.6%
Deprecation, Hardware	5.8%
Array, Conditional, Deprecation, Hardware	5.7%
Array, Deprecation, Hardware	5.7%
Array, Deprecation, Hardware, Type	5.7%
Conditional, Deprecation, Hardware, Type	5.7%
Conditional, Deprecation, Hardware	5.7%
Deprecation, Hardware, Type	5.7%
Pre-compilation, Type	5.7%

as type insatiability defects. Based on our findings, we conjecture that automated detection of anti-patterns, i.e., violation of recommended practices, can help developers to write better quality Julia code. Julia-based parsers, such as `CSTParser` (julia vscode, 2022), can be used for automated detection of Julia anti-patterns. Such anti-pattern detection tools could also be integrated as plugins in mainstream IDEs, such as Visual Studio Code.

Automated repair of Julia programs: Our findings provide the groundwork to repair defects automatically in Julia programs. One option is to leverage patterns from the defect fix examples for each defect category. Our dataset can be helpful in this regard as the dataset provides a mapping of what code elements are needed to fix defects in Julia programs. Researchers can leverage this mapping to extract patterns needed to generate necessary repairs.

Identified Defect Categories - Differences and Similarities: While certain categories described in Section 5 apply for generic software, we have identified other categories unique to Julia. Let us consider *pre-compilation defects*. For pre-compilation, Julia uses a specific directive, unique to the language itself. In the case of *polyglot defects*, the manifestation is entirely dependent on Julia-provided utilities, such as `ccall`. Similar argument applies for *world*

age defects as well. `World age` is unique to Julia's design and violation of this principle can lead to defects.

Findings reported in Section 5 also show similarities between identified defect categories for Julia and existing defect categories documented in prior work. One such category is *conditional defects*, which has been reported in generic defect taxonomies (Chillarege et al., 1992a; Seaman et al., 2008a), as well as in domain-specific software, such as for autonomous vehicles (Garcia et al., 2020), Puppet development (Rahman et al., 2020), and deep learning libraries (Islam et al., 2019). *Deprecation defects* are also commonplace: for example, deprecation is identified as a defect category for deep learning libraries, such as Keras and Tensorflow (Islam et al., 2019; Humbatova et al., 2020; Zhang et al., 2018a), as well as for Apache and Eclipse projects (Catolino et al., 2019). Zhang et al. (Zhang et al., 2018a) reported how changes in the Tensorflow API can lead to defects, which was further confirmed by Humbatova et al. (Humbatova et al., 2020), who documented the existence of 'deprecated API' defects for projects that use deep learning libraries, such as Keras and Tensorflow. *Hardware compatibility defects* have been documented by Makhshari and Mesbah (Makhshari and Mesbah, 2021) for IoT development, where they discussed how compatibility issues related to Raspberry Pi causes defects. We also have documented a compatibility-related defect in a Julia program when executed on a Raspberry Pi device in Section 5. In the case of *I/O operation defects*, prior work has documented evidence for generic (Seaman et al., 2008a) and domain-specific software projects, such as autonomous vehicles (Garcia et al., 2020). In the case of *type defects*, we observe commonalities with respect to regex defects and conversion defects. Use of wrong types that necessitate conversion to the correct type is common in deep learning-based projects (Humbatova et al., 2020). Regex defects are also commonplace in software engineering (Wang et al., 2020).

Developer Perception and Empirical Evidence: In an online forum (jul, 2020a), we observe developers to ask about defect categories in Julia programs: "*Defect types in Julia: what types of defects in Julia programs have you experienced?*" In response, forum participants mentioned type-related defects. Developer-reported perceptions related to defects in software can lack substantiation (Devanbu et al., 2016), which is subject to empirical validation. Along with type-related defects, we have identified 8 more defect categories, which further highlights the importance of our empirical study.

Our findings from Section 7 provide nuanced perceptions of developers related to the frequency and severity of identified defect categories. Based on developer response, type-related defects are most frequent, which is congruent with results presented in Table 3. Despite agreements with respect to frequency, survey respondents' perceptions related to severity are incongruent with that of our symptom analysis. For example, >90% survey respondents report deprecation and polyglot to have 'not at all'/'low' severity, even though these defect categories lead to crashes.

Documentation-related Recommendations: Our empirical study provides evidence that developers violate Julia-related best practices, such as writing

code that is compliant with type stability and world age. The Julia documentation can be improved so that these concepts are better disseminated with examples.

Application of Our Methodology: The major challenge for conducting this type of empirical study is the qualitative analysis portion, where raters with expertise on software defects need to individually inspect commits and issue reports. Another challenge is scalability: the more the commits and issues there are, the longer will take to conduct the analysis. In our case, the defect categorization process took 29 and 33 hours respectively, for the first and second author. Once these raters are identified, the methodology is generic enough to be adopted for any emerging programming language. Our defect categories can also be applied and extended for other emerging programming languages as well, as our paper provides definitions, examples, and symptoms for all defect categories.

Defect Frequency: Except for type-related defects, we observe majority of practitioners to perceive Julia-related defects to be infrequent and less severe. The frequency-related perceptions is congruent with empirical evidence as we identify 742 defects from 30,494 commits and 3,038 issue reports. One possible explanation for the infrequent defects is that as Julia is an emerging programming language, there is a lack of repositories that may have contributed to the reported perceptions and identified defect count. Another possible explanation is that our empirical study is reliant on reported defects as documented in commits and issue reports. It is possible that practitioners who maintain these repositories may have missed defects that they inadvertently introduced.

While we acknowledge that all of these explanations are possible, we still believe that our empirical study could be useful for the research community to conduct novel research, and for toolsmiths to develop tools that will aid practitioners who use Julia.

When applying qualitative analysis our focus was on deriving defect categories, where we derived each category based on the uniqueness for each category. While deriving these categories we do not consider the frequency of each category. As a result, in our categorization we have identified a defect category that includes only instance, namely world age.

Importance or Severity of Defect Categories: In our paper, we do not make any claim on the importance of each identified defect categories. Our survey analysis from Section 7 provided practitioner perception on the severity for each defect category, which can help researchers to contextualize the severity or importance of each identified defect categories.

Implications for Toolsmiths: We provide the following recommendation's based on our empirical analysis:

Deprecation repair tools: This category of tools will automatically detect deprecated syntax in Julia programs and repair detected deprecation instances. For example, this category of tools will automatically detect that `symbol` is deprecated, and repair automatically with `Symbol`.

Security tools: This category of tools will automatically detect security weaknesses in Julia programs (e.g. use of `unsafe_convert()`), and provide recommendations on how to fix them.
Polyglot tools: This category of tools will automatically generate code so that developers can easily interface Julia code with C and Fortran code
Detection tools for best practice violations: This category of tools will identify violations of Julia-related best practices automatically. Listing 19 shows evidence on how defects can be introduced into Julia programs because of violating Julia-related best practices. While the Julia community provides a set of recommended development practices (Jul, 2022d), developers can benefit from techniques, which can automatically identify violation of Julia best practices. For example, this category of tools will report where the Julia principle of type instability and world age is being violated in the code. Possible steps to develop these kinds of tools include but are not limited to:

- Curate Julia-related coding practices;
- Leverage code examples that map to each of the recommended practices;
- Generate rules by abstracting code patterns that are reflective of a violation for a certain recommended practice. For example, for `gc.enable()`, we observe that when the `enable()` function is called, then it violates the recommended practice of using `gc.@preserve()` instead of `gc.enable()`. This violation can be abstracted as $\neg gc.enable() \wedge gc.preserve()$, which in turn can be used as rule to inspect Julia programs that includes this violation;
- Detect instances of best practice violations with Julia-provided syntax analyzers, such as `CSTParser` Jul (2022d); and
- Apply information flow analysis techniques to mitigate false positive instances

Implications for Researchers: We list the following implications as future directions for researchers:

- Use our derived defect examples to generate example-guided repair techniques for Julia programs;
- Identify latent defects in Julia programs;
- Develop automated testing techniques for Julia programs;
- Gain an understanding of defects that appear for the Julia compiler; and
- Systematically investigate the feasibility of existing fuzzing techniques to detect defects in the Julia compiler.

9 Threats to Validity

In this section, we describe the limitations of our paper:
Conclusion Validity: The identified defect categories are prone to rater bias, which we mitigate by allocating multiple raters. Besides this, our analysis is limited as we determine the defect categories for Julia programs by mining

issue reports and commit messages, which may not provide full context. Furthermore, our keyword-based approach may might miss defect-related commits that do not include the keywords that we used. We mitigate this limitation by applying open coding on a sample of 378 commits, from where we do not identify new defect categories.

Internal Validity: Since the defect categories are identified based on the set of GitHub-related artifacts, as well as rater judgment, we cannot consider this list of categories to be comprehensive. We mitigate this limitation by systematically analyzing 30,494 commits and 3,038 issue reports. Our repository filtering criteria are based on heuristics, which may not identify all repositories with sufficient Julia source code files.

External Validity: Our findings may not generalize to OSS repositories that are not included in our analysis. Our findings may also not generalize for proprietary datasets. We mitigate these limitations by mining OSS Julia-related repositories hosted on GitHub.

10 Conclusion

Julia was designed to provide syntax similar to that of scripting languages, with the similar program execution speed of compiled languages that have low-level memory access. An empirical study, which systematically investigates defects in Julia programs can be beneficial for the Julia community as such a study can yield insights on why defects in Julia programs appear, and derive actionable recommendations to mitigate such defects. We have conducted an empirical study with 742 defects that appear in Julia programs by mining 112 OSS repositories. We identify 9 defect categories and 7 defect symptoms for the collected 742 defects. We observe certain categories, namely, polyglot, precompilation, and world age to be unique to Julia programs. Our findings also reveal that defects in Julia programs result in crashes and reduced program execution speed. Our findings provide groundwork on how to develop tools and documentation resources so that developers are well-equipped in developing Julia programs.

Conflict of Interests/Competing Interests The authors have no relevant financial or non-financial interests to disclose.

Data Availability Statements Dataset and source code used in our paper is publicly available online (Rahman, 2022).

Acknowledgements 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 the U.S. National Security Agency (NSA) Award # H98230-21-1-0175.

References

(2017) Julia joins petaflop club. https://www.hpcwire.com/off-the-wire/julia-joins-petaflop-club/
(2019) Julia: come for the syntax, stay for the speed. https://www.nature.com/articles/d41586-019-02310-3
(2020a) Bug types in Julia: what types of bugs in Julia programs have you experienced? https://discourse.julialang.org/t/bug-types-in-julia-what-types-of-bugs-in-julia-programs-have-you-experienced-are-they-different-from-bug-types-in-general-programming-languages-like-c-c/38640
(2020b) Programming languages: Developers reveal what they love and loathe, and what pays best. https://www.zdnet.com/article/programming-languages-developers-reveal-what-they-love-and-loathe-and-what-pays-best/
(2020c) Why julia is slowly replacing python in machine learning and data science. https://www.section.io/engineering-education/why-julia-is-slowly-replacing-python-for-machine-learning-and-data-science/
(2022a) Julia. https://juliacomputing.com/case-studies/celeste.html
(2022b) Julia. https://juliacomputing.com/case-studies/lanl/
(2022c) Julia. https://juliacomputing.com/case-studies/lincoln-labs/
(2022d) The Julia language. https://docs.julialang.org/en/v1/
(2022) The julia programming language. https://julialang.org/#
abelsiqueira (2015) https://gist.github.com/abelsiqueira/d4ca585c62204516bf37, [Online; accessed 19-Aug-2021]
Agrawal A, Rahman A, Krishna R, Sobran A, Menzies T (2018) We don't need another hero?: The impact of "heroes" on software development. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, New York, NY, USA, ICSE-SEIP '18, pp 245–253, DOI 10.1145/3183519.3183549, URL http://doi.acm.org/10.1145/3183519.3183549
amckay (2016) bug fix in markov chain simulation. https://github.com/QuantEcon/QuantEcon.jl/commit/097cf3, [Online; accessed 17-Feb-2022]
Antinyan V, Staron M, Sandberg A (2017) Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. Empirical Software Engineering 22(6):3057–3087
Beizer B (1984) Software system testing and quality assurance. Van Nostrand Reinhold Co.
Belyakova J, Chung B, Gelinas J, Nash J, Tate R, Vitek J (2020) World age in julia: Optimizing method dispatch in the presence of eval. Proc ACM Program Lang 4(OOPSLA), DOI 10.1145/3428275, URL https://doi.org/10.1145/3428275
Bezanson J, Chen J, Chung B, Karpinski S, Shah VB, Vitek J, Zoubritzky L (2018) Julia: Dynamism and performance reconciled by design. Proc ACM Program Lang 2(OOPSLA), DOI 10.1145/3276490, URL https://doi.org/10.1145/3276490

Carver JC (2009) First international workshop on software engineering for computational science engineering. Computing in Science Engineering 11(2):7–11, DOI 10.1109/MCSE.2009.30

Catolino G, Palomba F, Zaidman A, Ferrucci F (2019) Not all bugs are the same: Understanding, characterizing, and classifying bug types. Journal of Systems and Software 152:165 – 181, DOI https://doi.org/10.1016/j.jss.2019.03.002, URL http://www.sciencedirect.com/science/article/pii/S0164121219300536

Chen Z, Yao H, Lou Y, Cao Y, Liu Y, Wang H, Liu X (2021) An empirical study on deployment faults of deep learning based mobile applications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, pp 674–685

Chillarege R, Bhandari I, Chaar J, Halliday M, Moebus D, Ray B, Wong MY (1992a) Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering 18(11):943–956, DOI 10.1109/32.177364

Chillarege R, Bhandari I, Chaar J, Halliday M, Moebus D, Ray B, Wong MY (1992b) Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering 18(11):943–956, DOI 10.1109/32.177364

Churavy VVR (2019) Transparent distributed programming in Julia. PhD thesis, Massachusetts Institute of Technology, URL https://dspace.mit.edu/handle/1721.1/122755

Cinque M, Cotroneo D, Corte RD, Pecchia A (2014) Assessing direct monitoring techniques to analyze failures of critical industrial systems. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp 212–222, DOI 10.1109/ISSRE.2014.30

Devanbu P, Zimmermann T, Bird C (2016) Belief and evidence in empirical software engineering. In: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '16, pp 108–119, DOI 10.1145/2884781.2884812, URL http://doi.acm.org/10.1145/2884781.2884812

Di Franco A, Guo H, Rubio-González C (2017) A comprehensive study of real-world numerical bug characteristics. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 509–519

Drvi (2020) Fix inequality in reservoir sampling. https://github.com/joshday/OnlineStats.jl/commit/023df7, [Online; accessed 16-Feb-2022]

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting Empirical Methods for Software Engineering Research, Springer London, London, pp 285–311

eulerkochy (2020) Fix parametric initialisation of rbtree. https://github.com/JuliaCollections/DataStructures.jl/commit/b04a52, [Online; accessed 10-Feb-2022]

Friess E (2019) Scrum language use in a software engineering firm: An exploratory study. IEEE Transactions on Professional Communication

62(2):130–147, DOI 10.1109/TPC.2019.2911461
Gagniuc PA (2017) Markov chains: from theory to implementation and experimentation. John Wiley & Sons
Garcia J, Feng Y, Shen J, Almanee Y Sumaya Xia, Chen QA (2020) A comprehensive study of autonomous vehicle bugs. In: Proceedings of the 42nd International Conference on Software Engineering, ICSE '20, to appear
Gibson J (2017) The julia programming language: the future of scientific computing. APS pp L39–011, URL https://ui.adsabs.harvard.edu/abs/2017APS..DFDL39011G/abstract
github (2023) Blocking command line pushes that expose your personal email address. https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/, [Online; accessed 22-Feb-2023]
Gmys J, Carneiro T, Melab N, Talbi EG, Tuyttens D (2020) A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. Swarm and Evolutionary Computation p 100720, URL https://www.sciencedirect.com/science/article/abs/pii/S2210650220303734
Hickey G, Kipping C (1996) A multi-stage approach to the coding of data from open-ended questions. Nurse researcher 4(1):81–91
Howison J, Herbsleb JD (2011) Scientific software production: Incentives and collaboration. In: Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work, Association for Computing Machinery, New York, NY, USA, CSCW '11, p 513–522, DOI 10.1145/1958824.1958904, URL https://doi.org/10.1145/1958824.1958904
Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 1110–1121, DOI 10.1145/3377811.3380395, URL https://doi.org/10.1145/3377811.3380395
Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2019, p 510–520, DOI 10.1145/3338906.3338955, URL https://doi.org/10.1145/3338906.3338955
Januszek T, Pleszczyński M (2018) Comparative analysis of the efficiency of Julia language against the other classic programming languages. Silesian Journal of Pure and Applied Mathematics 8, URL https://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-c4339453-4519-4b92-a673-307638a50cb1
jayschwa (2014) Fix infinite recursion on 32-bit machines. https://github.com/JuliaGL/GLFW.jl/commit/89343b, [Online; accessed 19-Feb-2022]

Julia (2021) https://docs.julialang.org/en/v1/base/numbers/#Core.UInt64, [Online; accessed 29-Aug-2021]

julia (2021) Julia computing. https://juliacomputing.com/, [Online; accessed 11-Oct-2021]

JuliaLang (2017) https://github.com/JuliaLang/IJulia.jl/issues/573, [Online; accessed 10-Aug-2021]

JuliaLang (2021) Julialang/ijulia.jl. https://github.com/JuliaLang/IJulia.jl, [Online; accessed 10-Aug-2021]

JuliaLang/IJuliajl (2017) Pkd.build("ijulia") failing on arm. https://github.com/JuliaLang/IJulia.jl/issues/516, [Online; accessed 25-Aug-2021]

JuliaSmoothOptimizers (2019) Juliasmoothoptimizers/linearoperators.jl. https://github.com/JuliaSmoothOptimizers/LinearOperators.jl/issues/110, [Online; accessed 09-Aug-2021]

JuliaSmoothOptimizers/CUTEstjl (2015) Segmentation fault. https://github.com/JuliaSmoothOptimizers/CUTEst.jl/issues/45, [Online; accessed 19-Aug-2021]

Kitchenham BA, Pfleeger SL (2008) Personal Opinion Surveys, Springer London, London, pp 63–92. DOI 10.1007/978-1-84800-044-5$_3$, $URLhttps://doi.org/10.1007/978-1-84800-044-5\_3$

Krippendorff K (2018) Content analysis: An introduction to its methodology. Sage publications

Krippendorff K, Fleiss JL (1978) Reliability of binary attribute data

Krishna R, Agrawal A, Rahman A, Sobran A, Menzies T (2018) What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, New York, NY, USA, ICSE-SEIP '18, pp 306–315, DOI 10.1145/3183519.3183548, URL http://doi.acm.org/10.1145/3183519.3183548

Lattner C, Adve V (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, pp 75–86

Lombard M, Snyder-Duch J, Bracken CC (2010) Practical resources for assessing and reporting intercoder reliability in content analysis research projects

Makhshari A, Mesbah A (2021) Iot bugs and development challenges. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp 460–472, DOI 10.1109/ICSE43902.2021.00051

Mauny M, Vaugon B (2014) Nullable type inference. In: OCaml 2014-The OCaml Users and Developers Workshop

Merchant FA, Castleman KR (2005) 10.10 - computer-assisted microscopy. In: BOVIK A (ed) Handbook of Image and Video Processing (Second Edition), second edition edn, Communications, Networking and Multimedia, Academic Press, Burlington, pp 1311–XLIV, DOI https://doi.org/10.1016/B978-012119792-6/50136-4, URL https://www.sciencedirect.com/science/article/pii/B9780121197926501364

MikeInnes (2017) https://github.com/JunoLab/Juno.jl/commit/bd49b4, [Online; accessed 20-Feb-2022]

MOSEK (2018) fix: three number version. https://github.com/MOSEK/Mosek.jl/commit/3e9c63dab09, [Online; accessed 09-Aug-2021]

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating GitHub for engineered software projects. Empirical Software Engineering pp 1–35, DOI 10.1007/s10664-017-9512-6, URL http://dx.doi.org/10.1007/s10664-017-9512-6

Murphy J, Brady ET, Shamim SI, Rahman A (2020) A curated dataset of security defects in scientific software projects. In: Proceedings of the 7th Symposium on Hot Topics in the Science of Security, Association for Computing Machinery, New York, NY, USA, HotSoS '20, DOI 10.1145/3384217.3384218, URL https://doi.org/10.1145/3384217.3384218

pfitzseb (2017) https://github.com/JunoLab/Juno.jl/commit/865068, [Online; accessed 21-Feb-2022]

Poulding S, Feldt R (2017) Automated random testing in multiple dispatch languages. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp 333–344, DOI 10.1109/ICST.2017.37

Rahman A (2022) Verifiability package for paper. https://figshare.com/s/35d775572bb840ebd392, [Online; accessed 15-Mar-2022]

Rahman A, Agrawal A, Krishna R, Sobran A (2018) Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. In: Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, ACM, New York, NY, USA, SWAN 2018, pp 8–14, DOI 10.1145/3278142.3278149, URL http://doi.acm.org/10.1145/3278142.3278149

Rahman A, Farhana E, Parnin C, Williams L (2020) Gang of eight: A defect taxonomy for infrastructure as code scripts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 752–764, DOI 10.1145/3377811.3380409, URL https://doi.org/10.1145/3377811.3380409, pre-print: https://akondrahman.github.io/papers/icse20_acid.pdf

randyzwitch (2015) https://github.com/johnmyleswhite/Vega.jl/commit/9e3046, [Online; accessed 19-Feb-2022]

Raulamo-Jurvanen P, Hosio S, Mäntylä MV (2019) Practitioner evaluations on software testing tools. In: Proceedings of the Evaluation and Assessment on Software Engineering, Association for Computing Machinery, New York, NY, USA, EASE '19, p 57–66, DOI 10.1145/3319008.3319018, URL https://doi.org/10.1145/3319008.3319018

Saldaña J (2015) The coding manual for qualitative researchers. Sage

Satyanarayan A, Moritz D, Wongsuphasawat K, Heer J (2016) Vega-lite: A grammar of interactive graphics. IEEE transactions on visualization and computer graphics 23(1):341–350

Seaman CB, Shull F, Regardie M, Elbert D, Feldmann RL, Guo Y, Godfrey S (2008a) 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, Association for Computing Machinery, New York, NY, USA, ESEM '08, p 149–157, DOI 10.1145/1414004.1414030, URL https://doi.org/10.1145/1414004.1414030

Seaman CB, Shull F, Regardie M, Elbert D, Feldmann RL, Guo Y, Godfrey S (2008b) 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, Association for Computing Machinery, New York, NY, USA, ESEM '08, p 149–157, DOI 10.1145/1414004.1414030, URL https://doi.org/10.1145/1414004.1414030

simonster (2014) Fix type instability in read_array. https://github.com/JuliaIO/HDF5.jl/commit/ce2c44, [Online; accessed 20-Feb-2022]

Smith E, Loftin R, Murphy-Hill E, Bird C, Zimmermann T (2013) Improving developer participation rates in surveys. In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp 89–92, DOI 10.1109/CHASE.2013.6614738

Sweeney A, Greenwood KE, Williams S, Wykes T, Rose DS (2013) Hearing the voices of service user researchers in collaborative qualitative data analysis: the case for multiple coding. Health Expectations 16(4):e89–e99

Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empirical software engineering 19:1665–1705

tknopp (2018) fix sampling density function. https://github.com/JuliaMath/NFFT.jl/commit/fc791b, [Online; accessed 18-Feb-2022]

Vitter JS (1985) Random sampling with a reservoir. ACM Trans Math Softw 11(1):37–57, DOI 10.1145/3147.3165, URL https://doi.org/10.1145/3147.3165

julia vscode (2022) julia-vscode/cstparser.jl. https://github.com/julia-vscode/CSTParser.jl, [Online; accessed 22-Feb-2022]

Wang P, Brown C, Jennings JA, Stolee KT (2020) An empirical study on regular expression bugs. In: Proceedings of the 17th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '20, p 103–113, DOI 10.1145/3379597.3387464, URL https://doi.org/10.1145/3379597.3387464

WGS-TB (2021a) Update calling_functions.jl. https://github.com/WGS-TB/MentaLiST/commit/3f59f7b, [Online; accessed 11-Aug-2021]

WGS-TB (2021b) Wgs-tb/mentalist. https://github.com/WGS-TB/MentaLiST, [Online; accessed 11-Aug-2021]

yuyichao (2016) https://github.com/JuliaPy/PyCall.jl/commit/6ff741, [Online; accessed 17-Feb-2022]

Zappa Nardelli F, Belyakova J, Pelenitsyn A, Chung B, Bezanson J, Vitek J (2018) Julia subtyping: A rational reconstruction. Proc ACM Program Lang 2(OOPSLA), DOI 10.1145/3276483, URL https://doi.org/10.1145/3276483

Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018a) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, p 129–140, DOI 10.1145/3213846.3213866, URL https://doi.org/10.1145/3213846.3213866

Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018b) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, p 129–140, DOI 10.1145/3213846.3213866, URL https://doi.org/10.1145/3213846.3213866