Empirical Software Engineering (2022) 27:21
https://doi.org/10.1007/s10664-021-10033-1

# Demystifying regular expression bugs

## A comprehensive study on regular expression bug causes, fixes, and testing

Peipei Wang[1] · Chris Brown[2] · Jamie A. Jennings[3] · Kathryn T. Stolee[3]

Accepted: 1 July 2021Published online: 05 November 2021

**Abstract**
Regular expressions cause string-related bugs and open security vulnerabilities for DOS attacks. However, beyond ReDoS (Regular expression Denial of Service), little is known about the extent to which regular expression issues affect software development and how these issues are addressed in practice. We conduct an empirical study of 356 regex-related bugs from merged pull requests in Apache, Mozilla, Facebook, and Google GitHub repositories. We identify and classify the nature of the regular expression problems, the fixes, and the related changes in the test code. The most important findings in this paper are as follows: 1) incorrect regular expression semantics is the dominant root cause of regular expression bugs (165/356, 46.3%). The remaining root causes are incorrect API usage (9.3%) and other code issues that require regular expression changes in the fix (29.5%), 2) fixing regular expression bugs is nontrivial as it takes more time and more lines of code to fix them compared to the general pull requests, 3) most (51%) of the regex-related pull requests do not contain test code changes. Certain regex bug types (e.g., compile error, performance issues, regex representation) are less likely to include test code changes than others, and 4) the dominant type of test code changes in regex-related pull requests is test case addition (75%). The results of this study contribute to a broader understanding of the practical problems faced by developers when using, fixing, and testing regular expressions.

**Keywords** Regular expression bug characteristics · Pull requests · Bug fixes · Test code

## 1 Introduction

Regular expression research in software engineering has explored performance issues (Cody-Kenny et al. 2017; Wang et al. 2019), comprehension (Chapman et al. 2017),

Communicated by: Georgios Gousios and Sarah Nadi

This article belongs to the Topical Collection: *Mining Software Repositories (MSR)*

✉ Peipei Wang
peipei.wang@bytedance.com

Extended author information available on the last page of the article.

translation between languages (Davis et al. 2019; Davis et al. 2019), and test coverage (Wang and Stolee 2018). These efforts are motivated by the observation that regular expressions are pervasive in systems. For example, through the lens of GitHub issues, a simple search for "regex OR regular expression" yields 227,474 results (and growing), with 25% of those still being open. Regular expressions are reported to cause 37% of string-related bugs (Eghbali and Pradel 2020), and yet, regular expressions are poorly tested (Wang and Stolee 2018).

This work aims to uncover the nature of the issues that relate to regular expressions, and in particular, the nature of the issues that developers end up addressing. This work further explores the role of test code in bug fixing for regex-related bugs.

As a lens into issues developers face, we explore merged pull requests (PRs) related to regular expressions (*regex-related pull requests*). The assumption is that merged pull requests represent concerns in code that developers find worthy of addressing. We target large open-source projects – specifically Apache, Mozilla, Google, and Facebook – that use the pull request model for code contributions. This allows us to study the problem, solution, and discussions about regular expressions in multiple programming languages. Prior work suggests that there are significant differences in some regex characteristics across programming languages (Davis et al. 2019), and our findings echo this: we likewise find differences in bug characteristics across languages.

The study of bug descriptions and code changes in the regex-related pull requests leads to one of our main contributions: a classification of regex bugs addressed by developers. The distribution of regex bugs shows that developers write regular expressions that are too constrained three-times as often as they write regular expressions that are too relaxed. This has implications for test case generation research, indicating the importance of generating strings that are outside the regular expression language. While most of the related work on incorrect regex usage focuses on isolated regular expression strings and overlooks the context where regexes are used, this regex classification points out the importance of context and reveals that incorrectly using regex API also causes multiple regex-related problems.

As regular expressions are under-tested (Wang and Stolee 2018), we are interested in the impact of regular expression bugs on the testing effort. That is, are developers motivated to write tests if there is a bug in their regular expression? Thus, we explore the test code changes alongside the regex-related changes in PRs. From the test case changes, we can observe developers' behaviors on testing regular expressions, which gives hints on how to improve regular expression testing.

The contributions of this work include:

- The first comprehensive empirical study on regular expression bugs in real-world open-source projects across multiple languages.
- Identification of root causes and manifestations of 356 regular expression bugs in 350 merged pull requests related to regular expressions.
- Analysis of regular expression bug fix complexity and the connection between root causes and the changes in a fix.
- Ten common patterns in regular expression bug fixes.
- A quantitative and qualitative analysis of pull request test code changes, illustrating their relationship with various types of regex concerns.

This paper extends prior work (Wang et al. 2020) by studying the test code changes in the pull requests of regex-related bugs. Besides the contributions listed above regarding test code, we provide test code comparisons with other datasets and benchmarks, explanations for why many regex-related bugs do not have test code changes, and clues as to why it is hard to test regular expressions.

The rest of the paper is organized as follows. Section 2 proposes our research questions which are answered in Sections 4, 5, and 6, respectively. Section 3 shows how we collect and analyze data. Section 7 discusses our observations and the implications of this study. Section 8 presents threats to validity. Section 9 presents the related work and Section 10 concludes this paper.

## 2 Research Questions

The goal of this study is to understand the regular expression bugs developers address in practice. We obtain our data via purposely selected GitHub pull requests and carefully analyze these pull requests to achieve this goal. Specifically, this study asks and answers the following questions:

RQ1: *What are the characteristics of the problems being addressed in regex-related PRs?*
We use an open card sort to categorize the root causes of the problems that pull requests deal with. Three root causes emerge: 1) the regex itself; 2) regex API; and 3) other code. Within each type of root cause, we further characterize different manifestations of the addressed problem and provide more details about each manifestation (see Section 4).

RQ2: *What are the characteristics of the fixes applied to regex-related PRs?*
In analyzing the fixes in regex-related PRs, we measure fix complexity with four PR features proposed in prior work (Gousios and Zaidman 2014): 1) minutes between PR opening and merging; 2) the number of commits in the PR; 3) the number of lines changed in the fixes; and 4) the number of files touched in the fixes. We then zoom in to study the four types of regex-related code changes: 1) regex edit; 2) regex addition; 3) regex removal; and 4) API changes. For each PR root cause and manifestation, we identify the dominant type of change. Finally, we identify ten common fix patterns to fix either a regex bug or a regex API bug (see Section 5).

RQ3: *What are the characteristics of the test code in regex-related PRs?*
Through analyzing the test code involved in regex-related PRs, we analyze whether a pull request has test code changes along with the regex-related code changes. If the PR does contain test code changes, using the granularity of test cases, we classify the changes as: 1) test case edit; 2) test case addition; or 3) test case removal. The information we collected from the test code splits the dataset into two groups: PRs not containing test code changes through which we understand why test code changes are not involved in the fixes, and PRs containing test code changes through which we get the distribution of change types (see Section 6).

## 3 Study

This section describes the data collection process and analyses to address RQ1, RQ2, and RQ3.

### 3.1 Dataset

Our dataset is a sample of merged GitHub pull requests. We chose merged GitHub pull requests for two reasons: 1) our study is oriented towards the existing solutions of regular

```
1     gettext(format('Changes in {0} {1}',
2               this.app.trans[this.app.guid],
3   -           this.app.version.substring(0,1)))));
4   +           /\d+/.exec(this.app.version)))));
```

**Fig. 1** Example of Regex Addition from a pull request in JavaScript (mozilla/zamboni#442)

expression problems. Compared to GitHub issues, merged pull requests provide us with both the problem description and a solution; and 2) merged pull requests indicate the priority of the regular expression concerns and the feasibility to fix them, which are not always satisfied by GitHub issues since they may cover very general regular expression discussions or Q&As and thus do not embody a direct solution.

### 3.1.1 Artifact Collection

As we aim to focus on real resolutions to real bugs, we examined repositories from established organizations with relatively mature development processes and active projects. These repositories have many commits, contributors, and culture around pull request use. We targeted four large active GitHub organizations: Apache (2020), Mozilla (2020), Google (2020), and (2020). Using the GitHub GraphQL API (2020), we searched for merged pull requests[1] with “regular expression” or “regex” in the title or description with the last update time before February 1st, 2019. We selected only repositories that have Java, JavaScript, or Python as the primary language, as these are the three most popular programming languages used on GitHub (2014). This resulted in 664 merged pull requests from 195 GitHub repositories in the 4 organizations.

### 3.1.2 Pruning

We limited our focus to pull requests that are **regex-related**. A PR is called *regex-related* only if there are changes to a regular expression or a regular expression API method. In regex-related PRs, there is at least one regular expression that is added, removed, or edited, or there is at least one modification to regex APIs. For example, Fig. 1 shows an example of the regex `/\d+/` being added on line 4. We manually inspected the 664 merged PRs and identified 350 of them (52.7%) as regex-related PRs, resulting in 356 regex-related bugs in total (six PRs contained two bugs each).

### 3.1.3 Final Dataset Description

The final dataset of 350 regex-related PRs comes from 135 GitHub repositories. Of the 350 PRs, 86 are from Apache repositories, 162 are from Mozilla repositories, 66 are from Facebook repositories, and 36 are from Google repositories. Table 1 shows the languages and organizations for the 350 regex-related PRs and the 135 repositories. For example, there are 69 regex-related pull requests from 38 Apache repositories whose primary language is Java. When analyzing regex-related code changes, we considered the overall code

[1]While we avoid many perils of mining GitHub (Kalliamvakou et al. 2014) through our selection of organizations and projects (i.e., Perils II, III, IV, V, and VI), evaluating only merged pull requests is Peril VIII and thus a threat to validity, as discussed in Section 8.

**Table 1** The number of regex-related pull requests (repositories) selected from the four GitHub organizations

| | Apache | Mozilla | Facebook | Google | Total |
|---|---|---|---|---|---|
| Java | 69 (38) | 2 (2) | 0 (0) | 8 (6) | 79 (46) |
| JavaScript | 12 (7) | 70 (31) | 59 (9) | 7 (2) | 148 (49) |
| Python | 5 (3) | 90 (26) | 7 (3) | 21 (8) | 123 (40) |
| Total | 86 (48) | 162 (59) | 66 (12) | 36 (16) | 350 (135) |

differences before and after the PR, hence avoiding issues from reworked commits (Peril VII Kalliamvakou et al. 2014). Because a pull request can handle multiple independent regular expression problems, six PRs are split, creating a final dataset with 356 bugs addressed by pull requests, or *regex-related bugs*. Our final data are publicly available (Wang et al. 2020).

## 3.2 Terms and Definitions

In analyzing the regex-related bugs and their characteristics, we used the following definitions of semantics, behavior, smells, and bugs in the classification. These concepts are used throughout the rest of the paper.

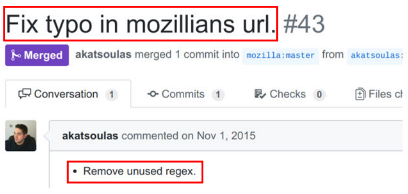
**Semantics and Behavior** The differences between code semantics and code behavior need to be clearly defined. The semantics of a formal language describes the meaning of syntactically valid statements and what they do. This applies to code semantics of a programming language as well as a regular expression.

Code semantics and regex semantics are measured by the correctness of the output given an input. Just as source code can look different and still have the same semantics (e.g., Pattern 3 in Table 5), regular expressions can look different but have the same semantics (Chapman et al. 2017) (e.g., `[a-zA-Z]5, and (?i)[a-z]5`).

Code behavior is a more general characterization, inclusive of semantics and side effects, such as execution time, memory consumption, and security vulnerabilities. The code before the change and after the change of Pattern 3 in Table 5 have different behaviors: one uses regex to detect the substring and the other uses string API. Although both are correct in semantics, the execution times differ.

**Smells and Bug** Code smells refer to code that has correct semantics but is flawed in another way, such as poor maintainability, performance, or readability (Chapman et al. 2017). Smells can be defined at different levels of granularity, such as *code smells* and *design smells*. Well-known code smells include *duplicated code* and *long method* (Fowler 2018). The well-known design smells include *spaghetti code* and *stovepip system* (Brown et al. 1998). For regular expressions, a *code smell* is a problem with a specific regular expression string or call site, whereas a *design smell* is a problem with how the regular expression is used in the code or which API was chosen.

In this paper, we use the term, *bad smell*, to describe the manifestation of bugs with correct regex semantics but could be improved in other ways. As a specific example, *performance bug* belongs to the category of *code smells* because the semantics are correct.



Fix typo in mozillians url. #43

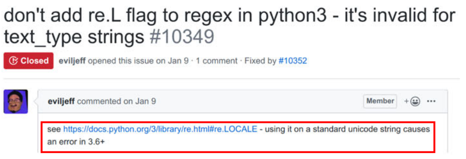
Merged akatsoulas merged 1 commit into mozilla:master from akatsoulas:

Conversation 1 Commits 1 Checks 0 Files ch

akatsoulas commented on Nov 1, 2015

- Remove unused regex.

(a) PR mozilla/feedthefox#43.



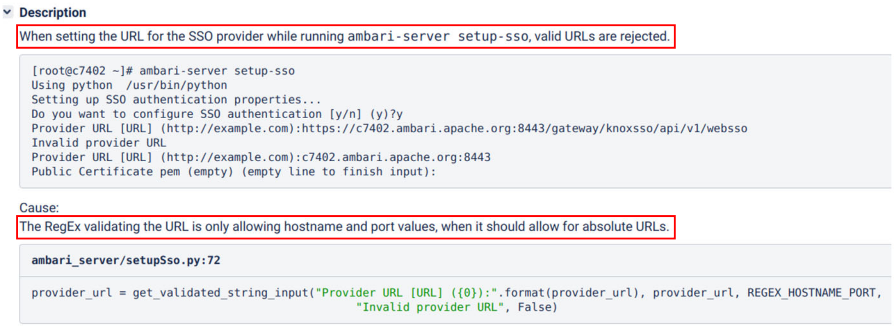
don't add re.L flag to regex in python3 - it's invalid for text_type strings #10349

Closed evilljeff opened this issue on Jan 9 · 1 comment · Fixed by #10352

eviljeff commented on Jan 9 Member

see https://docs.python.org/3/library/re.html#re.LOCALE - using it on a standard unicode string causes an error in 3.6+

(b) The GitHub issue for PR mozilla/addons-server#10352.



Description

When setting the URL for the SSO provider while running `ambari-server setup-sso`, valid URLs are rejected.

```
[root@c7402 ~]# ambari-server setup-sso
Using python  /usr/bin/python
Setting up SSO authentication properties...
Do you want to configure SSO authentication [y/n] (y)?y
Provider URL [URL] (http://example.com):https://c7402.ambari.apache.org:8443/gateway/knoxsso/api/v1/websso
Invalid provider URL
Provider URL [URL] (http://example.com):c7402.ambari.apache.org:8443
Public Certificate pem (empty) (empty line to finish input):
```

Cause:
The RegEx validating the URL is only allowing hostname and port values, when it should allow for absolute URLs.

```
ambari_server/setupSso.py:72

provider_url = get_validated_string_input("Provider URL [URL] ({0}):".format(provider_url), provider_url, REGEX_HOSTNAME_PORT,
                                          "Invalid provider URL", False)
```

(c) The JIRA issue for PR apache/ambari#760.

**Fig. 2** Examples to illustrate identifying problems addressed in pull requests

## 3.3 RQ1 Analysis: Bug Characteristics

With the 356 regex-related bugs, two authors performed an open card sort with two raters. The dataset is categorized in two dimensions, *root cause* and *manifestation*, based on the pull request description, comments, linked GitHub issues, or linked bug reports from other systems (e.g., JIRA, Bugzilla). Figure 2 shows three illustrative example PRs. In Fig. 2a, PR mozilla/feedthefox#43 addresses two problems. One is a typo of a variable shown in the title of this pull request, the other problem is an unused regex shown in the description of this pull request. We ignore the typo problem because the fix to the typo does not involve any regex or API changes. In the analysis of this PR, the fix is to remove the regular expression, and the problem it addresses is *unused regex* which is a type of *regex code smell*. Figure 2b shows a PR where the addressed problem is described in a separate GitHub issue, which identifies an error caused by an incorrect flag in the *regex API* with the manifestation of *exception handling*.[2] Figure 2c shows the JIRA bug report related to PR apache/ambari#760. Per the highlighted text, the problem being addressed in this PR is *incorrect regex semantics* because valid URLs are rejected and the scope of the regular expression needs to be expanded.

After card sorting was completed, eight manifestations of three root causes of regex-related bugs were identified. Four of the eight manifestations are further broken into

[2]The specific error message is “ValueError: cannot use LOCALE flag with a str pattern”. Since Python version 3.6, re.LOCALE can be used only with bytes patterns.

categories and sub-categories according to the common characteristics shared by the bugs. The hierarchy of the 356 regex-related bugs is presented in Table 2.

## 3.4 RQ2 Analysis: Fix Characteristics

To answer RQ2, we explored regex fix characteristics compared to general software bugs, the nature of the changes in the fixes, and identify common fix patterns.

### 3.4.1 Complexity of Regex-related PR Fixes

To understand if regex-related bugs are similar in complexity to other software bugs, we compared our regex-related PRs (*regexPRs*) with a public dataset of PRs from GitHub projects that use PRs in their development cycle (Gousios and Zaidman 2014) (*allPRs*). We selected four features from the prior work that represent the complexity of a fix or the complexity of reviewing a PR. To measure the complexity of reviewing the PR, we calculated the number of minutes from PR initialization to merging (*merge_mins*). To measure the complexity of fixes in PRs, we chose the number of commits (*num_commits*), the number of modified lines of code (*code_churn*), and the number of files changed (*files_changed*). Note that *code_churn* is a combined feature which is the sum of two originally proposed features, *src_churn*, the number of lines changed in source code, and *test_churn*, the number of lines changed in test code. This is because regular expressions are not only in source code but also in testing frameworks and configuration files, which makes it hard to distinguish the code of fixing a regex bug in production code from the one in the test code.

The metrics for bug fix complexity in our dataset (*regexPRs*) are obtained through the PyGithub (2020) library, which provides APIs to retrieve GitHub resources. The *allPRs* dataset (Gousios and Zaidman 2014) contains over 350,000 PRs; as a matter of fairness, we filtered out the unmerged pull requests and retained 300,600 merged ones for analysis. As our data do not follow a normal distribution (see *skewness* in Table 3), we used the non-parametric Mann-Whitney-Wilcoxon Test (2020) to investigate whether our dataset, *regexPRs*, and the *allPRs* dataset have the same distribution. These comparison results are presented in Table 3.

### 3.4.2 Changes to Regexes in PRs

We take into consideration four types of regex-related changes: 1) regular expression addition ($R_{add}$), 2) regular expression edit ($R_{edit}$), 3) regular expression removal ($R_{rm}$), and 4) regular expression API changes ($R_{API}$).

Before counting the number of regex-related changes, we first identified regular expressions used in the code. Because the regular expression is often represented as a string or a sequence of characters, we treated each *quoted* regex string as a normal string until we found it was parsed with regular expression syntax and a regular expression instance or object was created consequently. Strings wrapped by regular expression *delimiters* are straightforward and treated as regular expressions. For example, slash `/` in JavaScript is a regex delimiter. Hence `/\d+/` in Fig. 1 is identified as a regex.

A regular expression addition ($R_{add}$) is counted when the PR shows a new regular expression string. In the code snippet shown in Fig. 1, there is no regex string prior to the PR whereas line 4 introduces regular expression `/\d+/`.

```
   currentLine = subripData.readLine();
 - Matcher matcher = SUBRIP_TIMING_LINE.matcher(currentLine);
 - if (matcher.matches()) {
 + Matcher matcher = currentLine == null ? null :
       SUBRIP_TIMING_LINE.matcher(currentLine);
 + if (matcher != null && matcher.matches()) {
```

**Fig. 3** Example of Regex API changes from a pull request in Java (google/ExoPlayer#3185)

A regular expression edit ($R_{edit}$) is a content change to the regular expression string directly or indirectly used in regex API methods. These are the type of regular expression changes studied in prior work on regular expression evolution (Wang et al. 2019).

A regular expression removal ($R_{rm}$) is counted when a regular expression is removed entirely, and not just edited. A pull request could directly remove a regex object (e.g., mozilla/feedthefox#43) or replace the regex and the code where it is used with other types of code (e.g., google/graphicsfuzz#167).

A regular expression API change ($R_{API}$) encapsulates changes to the APIs being used statically and dynamically, as well as changes to the context of the API's use. This includes modifying the method itself on a call site and reducing the execution frequency of that call site. For modifying the API method, we counted only when the regex object is present both before and after the PR. Therefore, API methods introduced with $R_{add}$ or removed with $R_{rm}$ are excluded. Take Fig. 1 as an example. In this example, the method `exec` is added as the side-effect of adding the regex `/\d+/` and thus `exec` is not accounted as $R_{API}$. The modification to the method itself could be on its method name or arguments. If the modified argument is in the position for the regex string, it is not counted as an $R_{API}$ but as an $R_{edit}$. API changes also concern how the API methods are executed in run-time. For example, constructing regular expression objects statically rather than on-the-fly. The PR in Fig. 3 adds two checks of `null` object, one for the argument passed into `Pattern.matcher` and the other for the instance invoking `Matcher.matches`. Hence, it is counted to have two regular expression API changes. Another way of reducing call site execution frequency is to add guards (e.g., if-else statements) on the path of executing regular expression matching (e.g., mozilla/treeherder#61).

### 3.4.3 Recurring Patterns for Fixing Regular Expression Bugs

To find the common fix patterns, we examined the code changes in pull requests whose root cause is either regex or API (see Table 2). Since we are more interested in fixing regular expression bugs, the regex-related PRs caused by other code are out of scope for common fix patterns of regex bugs. Patterns are identified through bottom-up aggregation. Each regex-related change is regarded as a different pattern, and similar changes are grouped together. We chose ten explainable, recurring patterns to represent the fix strategies for common regular expression problems (See Table 5).

## 3.5 RQ3 Analysis: Test Code Characteristics

To answer RQ3, we identified the files used for test code[3] and then manually counted the test cases being changed in the PR. While rare, if a PR contained more than one regex-

[3] *Production code* is the part of the source code containing the logic of the software and runs in the production environment. *Test code* is the other part of the source code containing the tests which verify whether the production code exercises the expected logic.

related bug (six PRs in our dataset), we mapped the test code to the appropriate regular expression.

### 3.5.1 Identifying Test Code Files

Test code files are usually located under "**test**" (or "**tests**", "__**test**__") and "**spec**" (or "**specs**") directories (Gousios and Zaidman 2014; Munaiah et al. 2017). Following certain naming conventions, the test file names often contain the word "**test**" or "**spec**" as prefix or suffix (eg., pytest[4], javascript[5], Google java style[6]). Therefore, we regarded any files included in the PR which satisfy either of the two conditions as a candidate test code file.

We exclude test code files that do not have an impact on testing the regex-related code logic. For example, the only change in file *HadoopFsHelperTest.java* is added comments in PR apache/incubator-gobblin#63. Therefore, this file is not included when counting test code changes of this PR. Similar changes not considered in test code files include code formatting, variable renaming, and moving code location.

Note that a lack of test code changes in a PR does not mean test cases for the involved code do not exist. It just means that test code was not modified, added, or removed in the same PR.

### 3.5.2 Identifying Test Case Changes

Once test files are identified, we can investigate the test code changes in more detail. We chose to classify changes at the test case level of granularity because this granularity is comparable across programming languages and is not influenced by the organization of test cases into test files or classes. In prior literature, test cases (Pinto et al. 2012; Beller et al. 2017; Kochhar et al. 2013) are also used more often than test files and test classes (Zaidman et al. 2008).

The process for test case identification depended on the language and testing framework used. In the Python unit testing framework `unittest`[7] or `pytest`,[8] test method names start with the letters "**test**". In the Java Junit framework,[9] test methods are annotated with the `@Test` tag. In earlier Junit version 3, the unit test class is a subclass of `junit.framework.TestCase` with test methods prefixed with `test`. For behavior-driven testing frameworks (e.g., `mocha`,[10] `jasmine`,[11]) the `it` methods are regarded as the test case methods. Note that benchmark methods for testing regex-related code performance are also regarded as test cases. For the benchmark test files, each benchmark method is treated as a test case.

[4]https://docs.pytest.org/en/reorganize-docs/new-docs/user/naming_conventions.html

[5]https://stackoverflow.com/questions/49632743/what-is-the-convention-for-javascript-test-files

[6]https://google.github.io/styleguide/javaguide.html

[7]https://docs.python.org/3/library/unittest.html

[8]https://docs.pytest.org/en/stable/

[9]https://junit.org/junit5/

[10]https://mochajs.org/

[11]https://jasmine.github.io/

To identify test case changes, we looked for new test cases added into the test code files ($T_{add}$), test cases with modifications ($T_{edit}$), and test cases removed from the test files ($T_{rm}$). These are defined similarly to regex addition, edit, and removal (i.e., $R_{add}$, $R_{edit}$, and $R_{rm}$, respectively, see Section 3.4.2).

If the inputs for tested methods are not in the test method body, changes in test inputs are considered to be test case changes as well. The inputs may be located in the same test file as the test case or located in separate files. Taking PR mozilla/treeherder#181 as an example, file *test_tinderbox_print_parser.py* contains method `test_tinderbox_parser_output` where the test input source is `TINDERBOX_TEST_CASES`. Since the latter is added and it contains eight elements, $T_{add} = 8$ for this file with zero test case edits or removal. PR mozilla/treeherder#334 is another example where test case changes are not directly reflected in the test code file but in the input source files of the test code. The modified two JSON files in this PR are the expected results of test case methods `test_mochitest_fail` and `test_jetpack_fail` on file *tests/log_parser/test_job_artifact_builder.py*. Although there are no test code changes in these two methods, the expected results have changed. Therefore, this PR has $T_{edit} = 2$.

Besides the changes to test case methods and test case input sources, we evaluated the impacts of test fixtures on test cases as well. PR apache/ambari-metrics#8 shows the modified fixture method `setUp` contributes to test case edits of methods `testReporterStartStop` and `testMetricsExclusionPolicy`.

Similarly, we excluded code refactoring from the test code changes and also excluded test case changes that were made solely to pass the tests (Pinto et al. 2012). For example, in PR mozilla/fxa-auth-server#1743, the test file *sms.js* has a change from `/^US$/` to `[ 'US' ]` because in the configuration the format of `regions` has changed from a *RegExp* to *Array*.

## 4 RQ1 Results: Bug Categories

As is done in prior work on categorizing software bugs, we identify the *root cause* and *manifestation* of the bugs (Zhang et al. 2018; Lu et al. 2008; Di Franco et al. 2017; Selakovic and Pradel 2016; Tan et al. 2014). The root cause is the location in the source code wherein the problem lies. The manifestation is the impact of the bug on the code.

Among the 356 regex-related bugs, three root causes emerged: the **regex** itself (218, 61.2%), the **regex API** used (33, 9.3%), and **other code** (105, 29.5%), as shown in the *Root Cause* and *Count (%) in Root Cause* columns of Table 2. When the root cause is the *regex*, the regex itself causes an issue; examples include incorrect semantics[12], a compile error, or a code smell. When the *regex API* is the root cause, this means the API is deprecated, the wrong flags are used, the API call is unprotected from exceptions, or another issue related to the use of the API is present. When the root cause is *other code*, the regex-related changes are identified but the fault or root cause lies elsewhere in the code (i.e., the regex or API are modified in a fix, but are not the root cause of the issue).

[12] In this paper, we correct the naming of one of the manifestations from the original paper (Wang et al. 2020). In the original paper, we use the term, *incorrect behavior* but it should be *incorrect semantics*, per the definitions in Section 3.2.

**Table 2** The hierarchy for the 356 regex-related bugs including root causes, manifestation (manifest.), categories, and sub-categories (sub-category)

| Root Cause | Manifest. | Category | (Sub-Category) | Count (%) (sub-Cat.) | Count (%) Manifest. | Count (%) Root Cause |
|---|---|---|---|---|---|---|
| Regex | Incorrect Semantics | Rejecting valid strings | | 102 (61.8%) | 165 (75.7%) | 218 (61.2%) |
| | | Accepting invalid strings | | 36 (21.8%) | | |
| | | Rejecting valid and accepting invalid | | 17 (10.3%) | | |
| | | Incorrect extraction | | 9 (5.5%) | | |
| | | Unknown | | 1 (0.6%) | | |
| | Compile Error | | | | 8 (3.6%) | |
| | Bad Smells | Design Smells | Unnecessary regex | 11 (24.4%) | 45 (20.6%) | |
| | | | Other | 6 (13.3%) | | |
| | | Code Smells | Performance issues | 10 (22.2%) | | |
| | | | Regex representation | 10 (22.2%) | | |
| | | | Unused/duplicated regex | 8 (17.8%) | | |
| Regex API | Incorrect Computation | | | | 6 (22.2%) | 33 (9.3%) |
| | Bad Smells | Design Smells | Alternative regex API | 2 (7.4%) | 27 (81.8%) | |
| | | Code Smells | Unnecessary computation | 9 (33.3%) | | |
| | | | Exception handling | 8 (29.6%) | | |
| | | | Deprecated APIs | 5 (18.5%) | | |
| | | | Performance/Security | 3 (11.1%) | | |
| Other Code | New Feature | Data processing | | 22 (37.3%) | 59 (56.2%) | 105 (29.5%) |
| | | Regex-like implementation | | 19 (32.2%) | | |
| | | Regex configuration entry | | 18 (30.5%) | | |
| | Bad Smells | | | | 19 (18.1%) | |
| | Other Failures | | | | 27 (25.7%) | |
| Total | | | | | | 356 (100%) |

Each root cause is divided by the manifestation of the bug, which describes how the bug was observed (*Manifestation* and *Count (%) in Manifestation* columns of Table 2). For example, 45 PRs have *regex* as the root cause and manifest as a *bad smell*, representing 20.6% of the *regex* root cause. Categories and sub-categories are used to further subdivide the manifestations (*Category (Sub-Category)* and *Count (%) in Category* columns in Table 2). For example, 11 PRs have an *unnecessary regex*, representing 24.4% of the *bad smells* for the *regex* root cause.

Note that the manifestation of *bad smells* appears for each of the root causes. This is because the PRs will frequently identify a better way to accomplish a task through refactoring, which does not modify the semantics. These bad smells, in aggregate, account for 91 (25.6%) of the regex-related PR bugs. Next, we describe each root cause category.

## 4.1 Bugs Caused by Regexes Themselves

When the regex is an issue (218 PR bugs), we observed three manifestations: *incorrect semantics*, *compile error*, and *bad smells*.

### 4.1.1 Regex: Incorrect Semantics

*Incorrect Semantics* is the dominant manifestation for bugs with the regex as the root cause (75.7%, 165/218). Table 2 shows the four categories of this manifestation: rejecting valid strings, accepting invalid strings, rejecting valid and accepting invalid strings, and incorrect extraction. Rejecting valid strings represents 61.8% of the incorrect semantics bugs. This reinforces the observation that developers prefer to compose a conservative regex to an overly liberal one (Michael et al. 2019) and tend to expand the scope of regular expressions as software evolves (Wang et al. 2019).

Two factors frequently contribute to incorrect regex semantics. One factor is incorrect regex escaping, including not escaping characters and incorrectly escaping characters such as backslash (\) and forward slash (/). The other is changing requirements. When the inputs change and the regex is not updated, the regex semantics may become obsolete. For example, the regex in PR apache/cordova-ios#376 is used to look for the Apple OSX SDK version. The keyword preceding the version number used to be "OS X", but with Xcode 8+, it changed to "macOS". Since the search string changes, this regex must be updated as well. Other less common problems are related to case sensitivity, Unicode compatibility, misuses of quantifier greediness, and lack of anchors.

### 4.1.2 Regex: Compile Error

Eight pull requests fix regex *compile errors*. While the project code is compiled without errors, there could exist uncaught invalid regular expressions until runtime. For example, apache/nutch/#234 reports a runtime compile error caused by `File.separator` on Windows-based systems. Since \ is used for escaping other characters, this PR reports an uncaught *PatternSyntaxException*.

### 4.1.3 Regex: Bad Smells

The regex *bad smells* we observed can be divided into two categories, as shown in Table 2: *design smells*, such as whether to use regex solution or not, which data to use for validation, and what the matching data and non-matching data look like; and *code smells* referring to smells with the regex itself. Overall, 17 out of the regex bad smells are design smells and the other 28 are code smells.

Most design smells were sub-categorized as *unnecessary regex* (11/17). These PRs indicate that simpler solutions exist and a regex is not needed. For example, using a regex for string replacement is not necessary if the replaced string is a simple string literal in a fixed location (e.g., mozilla/Snappy-Symbolication-Server#23).

The *code smells* are roughly evenly distributed among three sub-categories. *Performance issues* means the execution of regex could be optimized for speed or memory consumption. For example, when the purpose of a regex is not to extract substrings from the data input, defined capturing groups in the regex is unnecessary since the captured values are saved in memory but not used in later code (e.g., apache/struts#156). Two of the performance issues are about regular expression complexity (i.e., ReDoS (Davis et al. 2018) vulnerability[13]). *Regex representation* means the regular expression string fails to satisfy certain unspecified requirements, such as using the raw string to describe regular expression in Python and following the eslint rule of "No-regex-spaces".[14] Six of the ten regex representation code smells can be detected by lint tools in Python and JavaScript. The other four PRs fix one issue of escape characters in regex strings and three issues of regex readability. Unlike incorrect regex escaping, which is one cause of the incorrect regex semantics manifestation, the escape characters in *regex representation* do not cause a semantic issue. *Unused/duplicated regex* refers to regexes in code that are no longer needed (7/8) or that are duplicated (1/8).

**Summary** Most instances of incorrect regular expression semantic occur when the regular expression is too conservative and needs to accept more strings. Compile errors occur in eight of the PRs, representing 2.2% of all regex-related PRs we studied; considering the severity of runtime compile errors in terms of disrupting the program execution, this is worth noting. Among design smells and code smells, 11 PRs identify the root cause as unnecessary regular expressions.

## 4.2 Bugs Caused by Regex APIs

Even with the correct regex, choosing the right API function is important, as is placing the API call in an appropriate location in the code. Bugs caused by regex APIs (33 PRs, 9.3%) refer to the incorrect regex API usage manifesting as either *incorrect computation* (6, 1.7%) or *bad smells* (27, 7.6%).

### 4.2.1 Regex API: Incorrect Computation

Six PRs were submitted because the API being used in the program produced incorrect results. For example, for a particular regular expression in (facebook/jest#3001), `RegExp.test(content)` has some unexpected behavior if it runs over the same string twice. This is because, in its context, the global matching flag `'g'` was used so the second call to this method starts matching from the position saved in the first call. In JavaScript this specifically affects *stateful* regex methods (i.e., `RegExp.test` and `RegExp.exec`).

[13]Since ReDoS cares about the time complexity of running the regular expression, we regard it as a performance issue.

[14]https://eslint.org/docs/2.0.0/rules/no-regex-spaces

Besides the stateful methods, other incorrect API usage leading to incorrect computation includes passing arguments into the wrong method, failing to process multi-line inputs, and enforcing matching from the beginning or to the end of an input string.

### 4.2.2 Regex API: Bad Smells

We found 27 PR bugs that stem from *bad smells* in using regex APIs. Table 2 shows the breakdown of the regex API bad smells. Two design smells are *alternative regex API* problems, such as deciding which regex library should be chosen to use (e.g., facebook/prepack#645). The other 25 (92.6%) are categorized as code smells.

*Unnecessary computation* was the root cause of nine PRs. In all cases, the issue is that the regex API is executed too many times and can be reduced. For example, on the code path where most of the jobs are a success, the regex parser for error messages should not be used unless the message indicates a job failure (e.g., mozilla/treeherder#61). This is considered a regex API issue because it pertains to how the API is used in the code. It is a code smell because the semantics are correct but some invocations of the regex API can be avoided for better performance. The frequency of this sub-category has implications for the impact regex API performance has on applications.

*Exception handling* refers to uncaught exceptions or errors in running regex methods. These represent issues with the regex APIs because developers did not account for the possible unexpected behaviors from executing a regex API. Examples include invalid regex syntax,[15] when the regex to compile is not hard-coded and unknown to the API method until runtime, invalid regex API method arguments (e.g., null values, unsupported regex flags), and invalid method returns (e.g., null values or incorrect return types).

*Deprecated APIs* means an obsolete regex library is being used or there were changes in the new version of a regex library. For example, the old regex library `org.apache.oro` is replaced with `java.util.regex` (apache/nutch#390) because `org.apache.oro` has been retired since 2010 and users are encouraged to use Java regex library instead.[16] Similarly, when the flags argument is deprecated[17] in JavaScript regex APIs, `input.replace('<', '&lt;', 'g')` has to be changed into `input.replace(/</g, '&lt;')` (mozilla/bugherder#26).

*Performance/Security* refers to a change in the API method due to performance or security concerns. For example, in JavaScript, developers found `regexp.test` to be more suitable than `str.match` because the former only returns a boolean value while the latter returns the matched results, which could create a leak of information to the external environment (mozilla/hubs#457).

**Summary** Understanding the regex API is as important as understanding the regex itself. PR bugs result from choosing the wrong API (6), using deprecated or updated APIs (5), or

[15] In Section 4.1.2 the regexes having compile errors are literal strings, hard-coded in the source code. In contrast, this section describes the regexes which are composed using variables that are passed into the regex API.

[16] https://jakarta.apache.org/oro/

[17] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace

improper exception handling (8). Additional PRs reduce the number of calls to the regex API in the interest of performance (9).

## 4.3 Bugs Caused by Other Code

In these bugs, regexes and their APIs are involved but are not the root causes of the bugs; the root cause is other code (105 PRs, 29.5%). Regexes may be changed in these pull requests, but the regex is part of the solution, not part of the problem. For example, to solve a filename comparison failure `filename=== 'jest.d.ts'` where the filename could be an absolute file path, a solution of regex matching is used to take the place (facebook/react#6804).

The manifestations of the regex-related PRs caused by code other than the regex or the regex APIs are categorized according to how regex-related changes are involved in the solution. A PR is categorized as a *new feature* if it implements new functionality or improves existing features (59 PRs). Note that we also regard feature improvement as a new feature. A PR is categorized as a *bad smell* if the regular expression is employed to refactor the source code and to remove the smells (19 PRs). A PR is categorized as *other failures* if it reports any other failure (27 PRs).

### 4.3.1 Other Code: New Feature

Regular expressions are often involved in the introduction of new features. For example, to prevent malicious injection into logs, a regex is added to sanitize log messages (https://github.com/apache/accumulo/pull/628), which means the root cause is unsanitized log messages, and sanitizing them is a new feature. Table 2 shows category the breakdown of the 59 PRs for new features.

*Data processing*, which accounts for 22 PRs, means the regular expression is added to process a specific type of data (e.g., mozilla/bugbug#65). *Regex configuration entry*, which accounts for 18 PRs, means the regex is user-provided so as to build regex-supported features satisfying different user needs (e.g., apache/openwhisk-utilities#16). *Regex-like implementation* adds new functionality for performing regular expression execution. It requires both a regex and an input string but provides some unique features. For example, a data query engine added query methods (e.g., regexp_matches) so that it can perform regex-like string searching in SQL queries (apache/drill#452).

### 4.3.2 Other Code: Bad Smells

When the root cause is a *bad smell*, the solution is a refactoring; the regex or its API is involved with the refactoring. For example, a switch statement of over 85 cases can be refactored into less than 20 cases through the use of regexes (apache/incubator-pinot#2894).

### 4.3.3 Other Code: Other Failures

Regular expressions can also be added when the existing solution in the code does not work. For example, a regex solution can be used as a fix when the code of identifying browser type fails to identify a newer version of the browser (mozilla/pdf.js#7800).

**Summary** Regexesv are involved in PRs even when the regex or its APIs are not the root cause.

**Table 3** Comparing selected features of regex-related PRs (*regexPRs*) to merged PRs (*allPRs*) from prior work (Gousios and Zaidman 2014)

| Feature | Meaning | Dataset | 5% | mean | median | 95% | skew | p-value |
|---|---|---|---|---|---|---|---|---|
| merge_mins | Time until | *allPRs* | 0 | 10,529 | 405 | 43,685 | 11 | – |
| | PR merge | *regexPRs* | 12 | 10,212 | 1,307 | 47,590 | 7 | 8.139e-13*** |
| num_commits | # of commits | *allPRs* | 1 | 4 | 1 | 11 | 17 | – |
| | in the PR | *regexPRs* | 1 | 3 | 1 | 8 | 8 | 0.3635 |
| code_churn | Changed LOC | *allPRs* | 0 | 324 | 15 | 1,047 | 32 | – |
| | in the PR | *regexPRs* | 2 | 616 | 27 | 786 | 18 | 1.075e-08*** |
| files_changed | # of changed | *allPRs* | 1 | 12 | 2 | 30 | 94 | – |
| | files in PR | *regexPRs* | 1 | 7 | 2 | 24 | 8 | 0.3068 |

*** p-value $<$ 0.001 when comparing *regexPRs* and *allPRs* for that feature using the Mann-Whitney-Wilcoxon test

## 5 RQ2 Results: Bug Fix Characteristics in Regex-related PRs

While RQ1 describes the regex-related PR bugs, RQ2 describes the associated fixes. We approach this from three perspectives: 1) the complexity of the fix, compared to general PRs; 2) the types of changes to the code; and 3) frequently recurring bug fix patterns.

### 5.1 Complexity of Regex-related PR Fixes

Given the prevalence and severity (Davis et al. 2018; Spishak et al. 2012) of bugs related to regular expressions, we are curious if there are properties of regex-related PRs that are different from most other PRs. We explore this idea by comparing characteristics of regex-related PRs to characteristics of PRs in a curated dataset from almost 900 GitHub projects without special filters on the pull requests (Gousios and Zaidman 2014). Table 3 shows the pull request feature distributions for our dataset (*regexPRs*) and the merged PRs from prior work (*allPRs*), as described in Section 3.4.1. We use non-parametric statistics since our data are non-normal (there is a long tail on the distributions). We compare the distributions of each feature across the datasets using a Mann-Whitney-Wilcoxen test (2020) of means. For each feature, we present the 5% percentile, mean, median, 95% percentile, and skewness score. The skewness score is calculated according to Pearson's moment coefficient of skewness (Pearson's moment coefficient of skewness | a blog on probability and statistics 2015).[18] For example, for the merged pull requests in *allPRs*, the median *num_commits* is 1 and the skewness is 16.75. Although the median number of commits is also 1 in *regexPRs*, the skewness of commits is only 7.97. This means the distribution of *num_commits* has a shorter tail in *regexPRs*, because of which the 95% percentile of *num_commits* in *regexPRs* is smaller than that in *allPRs*.

As shown in Table 3, *regexPRs* has less skewed distributions than *allPRs* on all features. Therefore, the characteristics of regex-related PRs are less asymmetric than general PRs. The Mann-Whitney-Wilcoxon tests between *regexPRs* and *allPRs* show that *regexPRs* take

[18]The skewness score for normal distributions is zero.

longer to merge (*merge_mins*) and involve more lines of code (*code_churn*), and these differences are significant at $\alpha = 0.001$. Based on these results, it appears that regex-related PRs are different than general PRs. However, further study is needed to understand if the differences observed here are due to presence of regexes rather than some other factor. For example, the sampled PRs come from different populations, with our *regexPRs* come from 135 repositories and *allPRs* come from 900 repositories.

**Summary** The fixes in regex-related PRs are significantly different from general PRs in that most regex-related PRs take a longer time to get merged and involve more lines of code.

## 5.2 Changes to Regexes in PRs

In regex-related PRs, we observed four types of changes: regex addition ($R_{add}$), edit ($R_{edit}$), or removal ($R_{rm}$), or a regex API is modified ($R_{API}$). Table 4 presents the distribution of regex changes over the 356 regex-related bugs with noted dominant type of regex changes. Across all root causes and manifestations, the most common change is an edit, as 52.8% (188/356) of the PRs contain one or more edit. Regexes were added in over twice the number of PRs (119) as they were removed (55). Regex API changes occurred in 59 (16.6%) of the PRs. Note that these numbers do not add up to 356 because a PR can have multiple types of changes (e.g., $R_{API}$ and $R_{edit}$); 14.9% (53/356) of the regex-related PRs involve more than one type of changes. Although $R_{edit}$ is the dominant type of regex-related change in our dataset, the number of $R_{edit}$ changes in those pull requests is usually one or two. In contrast, the average number of changes for $R_{API}$ is above seven. Next, we examined the fixes applied to each root cause.

**Table 4** Distribution of the four types of regex-related changes over different root causes and manifestations

| Root Cause | Manifestation (Category) | | #PR | $R_{add}$ | $R_{edit}$ | $R_{rm}$ | $R_{API}$ |
|---|---|---|---|---|---|---|---|
| Regex | Incorrect Semantics | | 165 | 22 (40) | 139 (236)● | 26 (48) | 12 (13) |
| | Compile Error | | 8 | 0 (0) | 7 (10)● | 1 (3) | 3 (3) |
| | Bad Smells | Design Smells | 17 | 4 (5) | 4 (9) | 12 (63)● | 3 (4) |
| | | Code Smells | 28 | 3 (3) | 20 (49)● | 8 (10) | 3 (5) |
| | Sum | | 218 | 29 (48) | 170 (304) | 47 (124) | 21 (25) |
| Regex API | Incorrect Computation | | 6 | 1 (1) | 1 (1) | 0 (0) | 6 (9)● |
| | Bad Smells | Design Smells | 2 | 0 (0) | 0 (0) | 0 (0) | 2 (2)● |
| | | Code Smells | 25 | 2 (8) | 3 (10) | 1 (25) | 23 (381)● |
| | Sum | | 33 | 3 (9) | 4 (11) | 1 (25) | 31 (392) |
| Other Code | New Feature | | 59 | 53 (110)● | 3 (4) | 0 (0) | 4 (4) |
| | Other Failures | | 27 | 23 (44)● | 6 (7) | 2 (4) | 3 (6) |
| | Bad Smells | | 19 | 11 (19)● | 5 (21) | 5 (20) | 0 (0) |
| | Sum | | 105 | 87 (173) | 14 (32) | 7 (24) | 7 (10) |
| Total | | | 356 | 119 (230) | 188 (347) | 55 (173) | 59 (427) |

A (B) means A PRs have B occurrences of the change, in total. ● indicates the dominant type of regex-related changes in the corresponding manifestation (or category) in each row

**Fixes for Regex Root Cause** When the regex is the root cause, 78.0% (170/218) of the PRs contain a regex edit. To fix design smells, however, regex removal is more common; as 11 of the 17 design smells PRs are related to unnecessary regexes (Table 2), removing the regex is a natural response.

We note that a regex edit is not always the solution, even when the regex itself is the root cause. For example, incorrect regex semantics could be fixed by replacing the regex with an existing parser (See Pattern 4 in Table 5). When incorrect regex semantics relates to the changed input data, the PR can either modify the regex or simply add a regex to the list of regexes (See Pattern 5 & 6 in Table 5). When the incorrect regex semantics is related to case sensitivity and Unicode characters, adding or modifying the regex flags in the regex API method can also be found together with regex edits (e.g., apache/beam#6092).

**Fixes for Regex API Root Cause** Most of the fixes for regex API issues involve changes to the API (78.8%, 26/33). Of all the API changes for all root causes (59 PRs, 427 instances), most fix deprecated APIs (71.2%, 304/427). However, multiple changes are sometimes required. For example, the PR mozilla/treeherder#198 handles an incorrect computation and contains an $R_{API}$ and an $R_{edit}$. While the fix moves the flag from `re.search` to `re.compile`, the regular expression `'.+ pgo(?:[ ]|-).+'` is optimized into a different representation `'.+ pgo[ -].+'`, which is a hidden regex representation code smell not mentioned in the PR description.

**Fixes for Other Root Causes** The majority (75%, 173/230) of $R_{add}$ edits come from the *other code* root cause. This is fitting as regexes are used in the solution for PRs in this category, but are not the cause of any issues.

**Summary** Suitably, each root cause has a common change type. When regexes are the problem, edits are the most common solution, unless it is a design smell that is resolved through removal. API problems involve API changes, and regexes are often added to solve problems caused by other code.

### 5.3 Recurring Patterns to Fix Regular Expression Bugs

Table 5 presents the ten recurring fix patterns we identified from the regex-related pull requests. Patterns 1-7 fix regex issues and patterns 8-10 fix regex API issues. The column *#PR* shows the number of pull requests that exhibit the pattern. However, this is not an indication of pattern frequency because a fix pattern can (and does) appear multiple times in the same PR. Pattern 7 is language-specific, but the rest are general enough to apply to the three languages: Python, JavaScript, and Java.

**Escaping Issues (Patterns 1 & 7)** Pattern 1 fixes incorrect regex semantics and compile errors that result from improper escaping, which we saw in Java, JavaScript, and Python. The domain knowledge required in Pattern 1 is to distinguish a regex meta-character from string escape character (e.g., `\b` can be a backspace or a regex word boundary) and from plain text (e.g., '`(`' can be a common left parenthesis or the starting anchor of a regex capturing group). Pattern 7 is specific to Python and can be used to distinguish regex meta-character escaping (e.g., `\.`) from string character escaping (e.g., `\n`).

**Table 5** Pattern 1-7 are to solve regex issues and Pattern 8-10 are to solve regex API issues. With the exception of Pattern 7 (as noted), each pattern can be applied to each of the languages studied: JavaScript, Python, and Java

| ID | #PR | Description with Example Before/After the PR | |
|---|---|---|---|
| 1 | 17 | Correctly escaping regex literals | |
| | | Before: | `regex="a.png"` |
| | | After: | `regex="a\.png"` |
| 2 | 17 | Extend or shrink the character class | |
| | | Before: | `value_regex = r'[_\w]+'` |
| | | After: | `value_regex = r'[_\-\w]+'` |
| 3 | 15 | Replace regex with string methods | |
| | | Before: | `if re.match(".*error.*",message):` |
| | | After: | `if "error" in message:` |
| 4 | 11 | Replace regex with existing parser | |
| | | Before: | `EMAIL_REGEX_PATTERN.matcher(email).matches();` |
| | | After: | `import javax.mail.internet.InternetAddress;` |
| | | | `InternetAddress emailAddr = new InternetAddress(email);` |
| | | | `emailAddr.validate();` |
| 5 | 10 | Add or remove a regex alternation | |
| | | Before: | `regex = "win32\|windows"` |
| | | After: | `regex = "wind32\|windows\|win64"` |
| 6 | 9 | Add or remove a regex to the regex list | |
| | | Before: | `'regexes': [` |
| | | | `    re.compile('Ubuntu HW 12.04 x64 .+')` |
| | | | `]` |
| | | After: | `'regexes': [` |
| | | | `    re.compile('Ubuntu (ASAN )?HW 12.04 x64 .+'),` |
| | | | `    re.compile('^Android 4\.2 x86 Emulator .+'),` |
| | | | `]` |
| 7 | 6 | Add or remove a regex to the regex representation; Language = {Python} | |
| | | Before: | `'pattern': '\d{1,2}/\d{1,2}` |
| | | After: | `'pattern': r'\d{1,2}/\d{1,2}'` |
| 8 | 5 | Checking null values for regex | |
| | | Before: | `Matcher matcher = regex.matcher(currentLine);` |
| | | After: | `Matcher matcher = currentLine == null ? null :` |
| | | | `↪ regex.matcher(currentLine);` |
| 9 | 4 | Regex static compilation | |
| | | Before: | `Strıng BLACKLIST = "...";` |
| | | | `boolean method(String name) {` |
| | | | `        return !(name.matches(BLACKLIST));` |
| | | | `}` |
| | | After: | `Pattern BLACKLIST = Pattern.compile("...");` |
| | | | `boolean methodE(String name) {` |
| | | | `        return !(BLACKLIST.matcher(name).matches());` |
| | | | `}` |
| 10 | 4 | Conditional checking before regex execution | |
| | | Before: | `Matcher m=Pattern.compile(regex).matcher(currentLine);` |
| | | After: | `if currentLine.contains("error"){` |
| | | | `    Matcher m =` |
| | | | `    ↪ Pattern.compile(regex).matcher(currentLine);` |
| | | | `}` |

**Regex Scope Issues (Patterns 2, 5 & 6)** Pattern 2 adds characters to a character class. Pattern 5 and Pattern 6 apply when additional alternatives are needed. When the strings within the regex are expressed in separate regular expressions, they can be combined in a single regex using an OR operator | or grouped into a set of regexes.

**Removing Regexes (Patterns 3 & 4)** Pattern 3 replaces the regex using string API functions while Pattern 4 replaces the regex solution with APIs provided in third-party libraries. The differences between Pattern 3 and Pattern 4 lie in the matching strings. Pattern 4 is used when the matching string has its own syntax grammar (e.g., email address, IP address, URL) and its dedicated parser. Pattern 3 is used when the use of string libraries is simpler or easier to understand than the regex implementation, but further research is needed to identify situations when a regex is better and when a string implementation is better.

**Exception Handling (Pattern 8)** Pattern 8 prevents null values from getting into or out of regex API methods. Another fix pattern for regex exception handling uses try-catch code blocks, but this can often be addressed by using smart editors to suggest exceptions to catch, so we omit it from the table.

**Unnecessary Computation (Patterns 9 & 10)** Pattern 9 avoids repeated regex compilation by pre-compiling regex objects and making the pre-compiled objects sharable among various functions. Pattern 10 reduces the execution frequency of regex methods by conditionally checking the input strings prior to the regex matching.

**Other Patterns** Other common patterns include transforming a regex character class into a regex shortcut, adding or removing regular expression anchors, changing regex API, splitting regular expressions apart or merging regular expressions together, or switching from capturing groups to non-capturing groups. For example, to support Unicode characters, simply adding the regex flag `re.UNICODE` to API functions (e.g., `re.findall`) is not enough. It is also necessary to change the character class from `[A-Za-Z0-9\']` to `[\w\']` because with the Unicode flag the shortcut `\w` supports Unicode characters but `[A-Za-Z0-9]` only support ASCII characters. To make the regex matching start at the beginning of a string, the developer can either change the API method from `[A-Za-Z0-9]` to `re.search(r"a+b"` to `re.match(r"a+b"` or add anchors to the regex so that it changes from `re.search(r"a+b")` to `re.search(r"^a+b")`. More patterns could be observed, but those presented in Table 5 represent common ones that are candidates for automation based on our careful exploration of the data.

**Summary** For a regex issue, there are often multiple fix patterns that can help, such as replacing a regex with string library operations or replacing it with external library calls. These patterns provide a first step toward understanding common regex-related code changes, which could enable automated program repair or other automated regex support.

## 6 RQ3 Results: Test Code Characteristics in Regex-related Bugs

In this section, we present the overview of test code changes of the 356 regex-related bugs in 350 GitHub pull requests.

**Table 6** The number of regex-related bugs w/o test code changes in the pull requests

| Total | Without test code change | With test code change | | |
|---|---|---|---|---|
| | | $T_{add}$ | $T_{edit}$ | $T_{rm}$ |
| 356 | 182 (51.12%) | 131 (75.29%) | 59 (33.91%) | 19 (10.92%) |
| | | | 174 (48.88%) | |

## 6.1 Overview

Table 6 shows the overview of test code changes among the 356 regex-related bugs. In total, 182 (51.12%) of them do not include test code changes in the pull requests while the other 174 (48.88%) contain changes in test code. In 171[19] bugs with test code changes, we find 984 impacted test cases, including 625 (63.52%) test cases added, 293 (29.78%) test cases edited, and 66 (6.71%) test cases removed.

For the purpose of comparison, we note the percentage of regex-related bugs with test code changes, 48.88%, is slightly higher than the reported 33% in another pull request study (Gousios et al. 2014). However, the differences in context matter as the prior work was conducted with GitHub projects sampled between 2012 and 2013, at a time when pull requests were newer to the development process. Differences in the types of contributions and the sizes of the changes can also influence developers' demand for test cases (Pham et al. 2013).

The dominant type of test case change is test code addition ($T_{add}$) as over 75% of bugs with test code changes contain added test cases (Table 6). On average, 4.77 test cases are added, 4.97 test cases are edited, and 3.47 test cases are deleted. The comparison of our dataset with other studies on test code changes (Pinto et al. 2012) reveals that the regex-related bugs with test code changes have a higher percentage of test case addition and a lower percentage of test removal. The percentages of $T_{add}$, $T_{edit}$, and $T_{rm}$ reported in (Pinto et al. 2012) are 56%, 29%, and 15%. Those percentages in regex-related PRs are 64%, 30%, 7%. The study on test case changes (Pinto et al. 2012) finds that new test cases are added to exercise changed code and to validate bug fixes and new functionalities while test cases are removed because they are obsolete. The slightly higher percentage of $T_{rm}$ might be the result of regex removal in the pull requests.

**Summary** Nearly 49% of regex-related bugs have test code changes. The dominant test code change type is test code addition which occurs in 75% of the regex-related bugs having test code change.

## 6.2 Regex-related Bugs without Test Code Changes

The analysis on regex-related bugs without updated test code seeks to understand the reason some bugs do not have test code changes. We first look at the relationship between the root cause and the test code change among the 356 bugs.

We observe that PRs related to certain types of regex-related bugs include few test code changes. As shown in Table 7, PRs for solving regex *compile errors* do not involve any

[19]We lost the data for three bugs between the original paper (Wang et al. 2020) and this analysis of the test code. We know there were test code changes, but the details are no longer available.

test code changes. Since regex compile errors will break all test cases involving the regular expression, often additional tests are not needed. It can also be the case that testing costs outweigh the perceived benefit. For example, the regex edit in PR apache/nutch#234 only impacts one line of code. However, the bug only occurs on Windows-based systems, which would require a specific test environment to be configured.

We also notice that PRs for solving performance-related bugs often do not have test code changes; these fall under the *bad smells* category with *regex* as the root cause. Seven of the ten bugs that manifested as *performance issues* contain no test code changes in the corresponding pull requests. The other three bugs do find test code changes; they either add test cases to test for regex hang bugs or adapt existing test cases to reflect the regex changes in the source code. None of the three involve test code changes to measure performance.

PRs for regex code smell *regex representation* also do not often contain test code changes as 80% (8/10) of such bugs are addressed in pull request without test changes. For the other two bugs having test changes, test cases are added because there did not exist tests for the changed regex prior to the PR; test cases are edited as new test scenarios are added to test the same functionality.

Besides the regex-related bug types, the location of the regular expression is another reason for bugs are fixed without test changes in pull requests. The regex change may not be considered as necessary to test if it is irrelevant to the software logic. For example, regexes being changed in PR google/openhtf#347 are used in the "pylint" tool and not in the source code. In PR mozilla/amo-validator#520, the modified regexes interweave with only the test code to assist running test cases.

Not having test code changes does not mean the test code does not exist. Therefore, test code changes may not be necessary when existing test cases are sufficient to describe the expected behaviors, especially when the regex-related change does not impact its matching behavior. For example, the bug described in PR apache/cordova-ios#376 is revealed by the test code. Also, for the regex API change of converting regex compilation to static (Patterns 9 in Section 5.3), running existing test code is enough to make sure it does not impact code functionality but only code execution time.

**Summary** Certain types of regex-related bugs (compile errors, performance issues, regex representation) rarely have test code changes in the pull requests. The location of the regular

**Table 7** The distribution of test code changes among the 356 regex-related bugs in each root cause and manifestation

| Root Cause | Manifestation | $T_{add}$ | $T_{edit}$ | $T_{rm}$ | Combined |
|---|---|---|---|---|---|
| Regex | Incorrect Semantics | 53 | 32 | 10 | 79/165 (47.88%) |
| | Compile Error | 0 | 0 | 0 | 0/8 ( 0.00%) |
| | Bad Smells | 10 | 7 | 2 | 16/45 (35.56%) |
| Regex | Incorrect Computation | 2 | 2 | 0 | 4/6 (66.67%) |
| API | Bad Smells | 6 | 1 | 1 | 7/27 (25.93%) |
| Other | New Feature | 40 | 9 | 2 | 44/59 (74.58%) |
| Code | Bad Smells | 11 | 3 | 3 | 13/19 (68.42%) |
| | Other Failures | 9 | 5 | 1 | 11/27 (40.74%) |
| Total | | 131 | 59 | 19 | 174/356 (48.88%) |

expression and the test code prior to the PR also play a role for developers not to make test code changes.

## 6.3 Regex-related Bugs with Test Code Changes

To understand the test case changes that go along with fixes for the regex-related bugs in same pull requests, we first look at the number of changed test cases, then check its relationship with the regex-related bugs and with the regex changes.

Table 8 presents the distribution of the number of changed test cases in the pull requests of 171 regex-related bugs. The columns $T_{add}$, $T_{edit}$, $T_{rm}$ shows the number of added, edited, and deleted test cases in the PR, separately. The column *combined* shows the total number of the changed test cases regardless of the test case change type. Note that we have in total 174 regex-related bugs that have test code changes in pull requests but 3 are excluded since we cannot count their test case changes.

While 75% of regex-related bugs have the test case changes less or equal to 5, we notice that the pull requests for some bugs have significant test case changes. We examined the six PRs which have more than 20 test case changes. Among them, three of them modify the list of test inputs that feeds into the test cases, two have at least two completely new test code files, and one modifies all test input files of a certain type.

Referring to Table 7, *new feature* caused by *other code* has the highest percentage of containing test code changes (44/59; 74.58%) and the highest percentage of added test cases (40/44; 90.91%), followed by test code changes for *bad smells* of *other code* (i.e., 74.58% (13/19) and 84.62% (11/13). This is in line with a prior study on test code evolution (Pinto et al. 2012) that shows 69% of the added tests are added to validate newly added code. For these two types of regex-related bugs, it is highly likely that new code is added into the program since regexes are used for either implementing new features or code refactoring. Therefore, the majority of the tests for these two types are most likely added to validate the new code as well. Actually, tests are even requested by PR reviewers before merging it into the code repository (e.g., PR apache/beam#5528).

Although $T_{add}$ is the dominant type of test code changes, bugs caused by *regex* stand out as they consist of a more balanced distribution of test code changes. The percentage of $T_{add}$ (66.32%; 63/95) is smaller than the average 75.29%; the percentage of $T_{edit}$ (41.05%; 39/95) is higher than the average 33.91%; the percentage of $T_{rm}$ (12.63%, 12/95) is also higher than the average 10.92%. One possible explanation is that the non-matching behaviors of regexes are not often tested (Wang and Stolee 2018). Therefore, when an invalid string is accepted, developers have the option to delete the invalid string or to edit the string into a

**Table 8** The distribution of test case changes among the 171 regex-related bugs with test code changes

| | $T_{add}$ | $T_{edit}$ | $T_{rm}$ | Combined |
|---|---|---|---|---|
| count | 131 | 59 | 19 | 171 |
| mean | 4.77 | 4.97 | 3.47 | 5.75 |
| min | 1.00 | 1.00 | 1.00 | 1.00 |
| 25% | 1.00 | 1.00 | 1.00 | 1.00 |
| 50% | 2.00 | 2.00 | 2.00 | 2.00 |
| 75% | 5.00 | 3.00 | 3.50 | 5.00 |
| max | 50.00 | 166.00 | 13.00 | 167.00 |

valid one. However, we have not found a plausible explanation for bugs causing regex *bad smells* due to the small number of pull requests (16) and the diversity of this bug type.

**Summary** 75% of the regex-related bugs having test code changes contain no more than five changed test cases. Although test case addition is the dominant test change type across bugs with different root causes, not all test cases are added to validate bug fixes. Depending on the type of the regex-related bug, test code changes are commonly used to validate new functionality, code refactoring, and bug fixes.

## 7 Discussion and Future Work

We began this work to gain a better understanding of the issues developers face when working with regular expressions, and the lens we chose is the pull request. Here, we discuss our high-level observations, implications, and future work possibilities. Based on our analysis of the data, the following observations stand out:

**Differences across programming languages** Prior work shows that the regular expression representations have significant differences across programming languages (Davis et al. 2019) and porting regular expressions causes semantic and performance differences (Davis et al. 2019). During our analysis of regex bugs, we saw that some regex bugs are closely related to a particular program language. The incorrect computation or incorrect regex semantics caused by stateful methods occur only in JavaScript. The regex API code smell of *Performance/Security* occurs in JavaScript and Python, but not Java (Section 4.2.2). The language version also has an impact on regex bugs by changing flags (e.g., `re.L` is no longer supported after Python 3), deprecating APIs, and changing performance.

**Regex issues when represented as string literals** When a regular expression is represented as a quoted string literal, it can be tricky to get right. Regexes use backslashes for shortcuts (e.g., `\d`) and to convert meta-characters to plain characters (e.g., `a\.png`). However, backslashes themselves need to be escaped to make a valid string sequence. The complicated escaping process and the different escaping character support in different languages make regular expression escaping fragile (see Pattern 1 in Table 5).

**To regex or not to regex** Our study found 15 PRs of replacing regex with string operations and 9 PRs of replacing string operations with regular expressions. When other code is the root cause of the issue, regexes are added in 82.9% (87/105) of the PRs. The problem of when regular expressions should and should not be used (Regex use vs. regex abuse 2005; Replacing a complex regular expression with a simple parser 2017; When you should NOT use Regular Expressions? 2011) is also discussed in the PRs. One PR discussion sets a boundary for when regexes should be used: "*If the data and the comparison only require you to test for equality, then I'd try to use an Array. If whatever I'm testing can't use equality then I'd use a RegExp.*" (mozilla/fxa-auth-server#1743). This problem is regarded to be one of the difficulties of regex programming (Michael et al. 2019). Further research efforts are needed to better understand when to use and when not to use regexes.

**Regex usage context matters** In this paper, we found that regex errors go beyond just composing the intended regex. The issues we observed also include incorrect usage of regular expression APIs (Section 4.2), improper exception handling (Section 4.2.2), and unreadable or inefficient regexes (Section 4.2.2). Thus, it is important to consider regexes in their context when proposing solutions to support developers. Online tools, which developers report to use for regex composition and testing (Chapman and Stolee 2016), cannot determine if a regex is compiled too often (Pattern 9, Table 5), if a string library would be more appropriate (Pattern 3, Table 5), or if a meta-character should be escaped (Pattern 1, Table 5). While helpful for understanding matching behavior, developers could benefit from tool support within the IDE that can consider the context.

**Regex performance is about more than regex complexity** Prior work on regexes and ReDoS (Davis et al. 2018; Staicu and Pradel 2018; Wüstholz et al. 2017; Shen et al. 2018) focuses on the complexity of executing a regular expression. In the PRs we studied, developers demonstrated an interest in optimizing regex execution by refactoring the surrounding code (e.g., adding conditional or null checking, Patterns 8 & 10, Table 5) or by fine tuning the features in the regular expression representation such as changing capturing groups to non-capturing groups (e.g., apache/nutch#432). There are several regex optimization tools (regexp-tree - npm 2021; Perl-compatible regular expression optimizer 2021) but the optimizations are primarily for the purpose of readability. In fact, the optimization of replacing `[0-9]` with `\d` could yield worse performance when the encoding is not ASCII (Stack overflow 2021). Removing capturing groups while ignoring that those capturing groups are used to extract substrings could cause semantic errors. For automated performance support, the context of regular expression usage would help developers identify these inefficiencies sooner.

**Testing regexes is uncommon and testing regex performance is rare** Prior work on regex testing (Wang and Stolee 2018) shows that only 17% regular expressions are tested. The PRs reveal that test code is not typically committed with regex changes; only 48.88% of pull requests addressing regex-related bugs include test code changes. The percentage of regex-related bugs with test changes in the same pull requests is much lower than that reported in other bug benchmarks (Madeiral et al. 2019; Gyimesi et al. 2019).[20] While real-world regex bug benchmarks for regex testing and repairing techniques would be beneficial, constructing such benchmarks could be more challenging given the low frequency of regex-related tests (Wang and Stolee 2018) and the low frequency of bug fixes containing changed tests.

There are very few test case changes regarding the performance impact from regex-related changes. For PRs addressing code smells related to performance, the optimization has either no demonstration, is made based on other resources, or is demonstrated by the program execution time prior to and after the PR. For example, PR apache/cxf#479 makes the regex changes based on a StackOverflow question and provides that as evidence in the bug description. A specialized testing benchmark for regular expressions could help developers make better regex usage optimization.

Note that we do observe some pull requests include performance information. For example, PR apache/druid#3642 containing measurements of performance using JMH benchmarks.[21] PR apache/cxf#479 also contains a benchmark to compare the execution

[20]The former reports 94% of found bug-fixing patches contain test cases and the latter reports only four patches with no tests changed.

[21]https://openjdk.java.net/projects/code-tools/jmh/

time of two Java methods (i.e., `String.split` and `Pattern.split`). However, the location is the pull request description instead of the test code. PR google/grr#131 and PR mozilla/treeherder#60 report the performance differences before and after the regex modification in the description but with no test code changes.

Through this exploration of test code, we reflect that the regex testing statistics from our prior work (Wang and Stolee 2018) may be artificially low due to feasibility issues. Not all regexes can be tested in context. For example, the regular expressions written in configuration files (e.g., mozilla/amo-validator#520) or the regular expressions causing compile errors (e.g., apache/nutch#234). In that case, it is important to ensure the regexes are not malicious and do not cause significant system slowdown.

**Summary** Each of these observations opens the door for further research. Our sample of PRs was not large, but the analysis was in-depth. Opportunities for further, automated exploration and further, automated support have been identified.

## 8 Threats to validity

**Internal Validity** Pull requests are used to propose both features and fixes. There is a chance that a few regex-related pull requests are actually feature requests. Although we could filter out such pull requests using labels, the majority of pull requests are not labelled (Yu et al. 2018). For the labelled ones, previous study haven shown that misclassification occurs frequently between bugs and non-bugs (Pingclasai et al. 2013). For these reasons, we did not distinguish the pull requests of new features during artifact collection. Instead, we classify them during the analysis of bug characteristics.

We manually labeled the pull requests using two authors as raters. To reduce misclassifications, all disagreements were thoroughly discussed with a third author.

We used GitHub's merge status in selecting PRs, which poses a threat to validity (Kalliamvakou et al. 2014). Additional pull requests may have been merged, and the existence of such pull requests would affect our results if they substantially differ from the ones merged via GitHub. Additionally, changes that precede or follow the PR but are not linked to the PR were not analyzed.

Based on the convention that code changes should be accompanied by tests (eg., react[22], google/guava[23], apache/airflow[24]), we only examined the test code before the pull requests and the test code changes in the pull request. As pointed out by Zaidman et al. (2008), in open-source software systems, test code and production code can evolve synchronously following the test-driven development instructions. However, they may also have periods of pure testing and pure development. Our observation and conclusion may not hold if the test code for the regex changes are made in a separate commit or pull request posterior to the PR making regex changes.

We use heuristics to detect test code files under certain directories. This might not identify all test code files related to the pull requests. However, we need to take these heuristics in order to evaluate the thousands of changed files in the pull requests. In analyzing pull requests with test code changes, we focus on whether test code changes come along with

[22]https://reactjs.org/docs/how-to-contribute.html
[23]https://github.com/google/guava/blob/master/CONTRIBUTING.md
[24]https://github.com/apache/airflow/blob/main/CONTRIBUTING.rst

the bug fixes or not as it is a convention that when programmers fix a bug they often modify the test code to reproduce the bug (Zhong and Su 2015). We include cases where test code changes committed separately earlier than the corresponding pull request. For the cases test code changes committed later than the pull requests, there is no clue when the test code changes would be committed. Therefore, we have to treat such cases same as the other pull requests not containing test code changes.

**Construct Validity** Among the 16 PR features (Gousios and Zaidman 2014), we only select four of them to evaluate RQ2. The comparison between *regexPRs* and the *allPRs* dataset may not hold on the other features.

When comparing *regexPRs* and *allPRs*, we observed that 8 PRs in *regexPRs* are present in *allPRs*. We believe the impact is minimal, as there are over 800x more PRs in the *allPRs* dataset.

Where appropriate, we connected our results to prior work on regular expressions to reduce mono-method bias.

**External Validity** The PRs were sampled on February 1, 2019, and thus reflect the PRs available up to a specific date and time. Although we believe the PRs we studied cover the majority of concerns about regular expressions, the distribution of those concerns may vary in data collected with different approaches.

We analyzed 356 bugs from merged PRs in 4 organizations, which may not be representative of all regex-related bugs. These PRs are in three languages, which may not generalize. The dataset is from public GitHub repositories, which may not generalize to projects hosted elsewhere or private repositories. However, we did not observe any important differences in PRs between the selected organizations. Their distributions of root causes and manifestations, are not statistically different from one another, suggesting (though not proving) generalizability.

We split six PRs into multiple bugs because the issues were independent. This has a subtle impact on the generalizability of the results to other sets of regex-related PRs.

## 9 Related Work

This work is mostly related to research on regular expressions in software engineering. The methodology is most related to research on software bugs and classification.

**Regular Expressions in SE** Empirical research on regular expressions in software engineering is emerging (e.g., (Chapman and Stolee 2016; Chapman et al. 2017; Davis et al. 2018; Wang and Stolee 2018; Wang et al. 2019; Davis et al. 2019; Davis et al. 2019; Park et al. 2019; Zhong et al. 2018; Chen et al. 2020)). Previous research focuses on regex feature usage in one language (Chapman and Stolee 2016) and later on comparing regex characteristics across languages (Davis et al. 2019; Davis et al. 2019), with a specific focus on portability issues (Davis et al. 2019). Previous research also explores regex characteristics (e.g., size, features) at a moment in time (Chapman and Stolee 2016; Davis et al. 2019), or, more similar to this work, on changes to the characteristics over time through the lens of commit history (Wang et al. 2019). Another dimension is context: some regular expression studies extract regexes from their context for analysis (e.g., (Chapman et al. 2017; Wang et al. 2019)), but others consider the execution environment (e.g., to measure test coverage (Wang and Stolee 2018) or identify actual ReDoS issues (Davis et al. 2018;

Staicu and Pradel 2018; Wüstholz et al. 2017; Shen et al. 2018)). Prior work has also shown researchers' interest in regular expression synthesis with example-based (Bartoli et al. 2016; Ye et al. 2020), NL-based (Zhong et al. 2018; Park et al. 2019; Locascio et al. 2016), and multi-modal (Chen et al. 2020) techniques. In this work, we analyzed regexes in multiple languages using the context from PRs, which is not available through commit history alone, in addition to properties of the regex itself.

Regex comprehension has been studied using controlled experiments (Chapman et al. 2017) and composition strategies have been studied using observational studies of developers (Bai et al. 2019). This work is complementary to work in regex comprehension, as regex representation code smells were found in this work. These are the byproduct, in part, of regex readability issues (Section 4.1.3).

Complementary to our efforts here, prior work identifies the presence of ReDoS vulnerabilities in thousands of JavaScript and Python modules (Davis et al. 2018; Staicu and Pradel 2018; Davis et al. 2021). While the prior work (Davis et al. 2018) took a deep dive into a particular type of vulnerability, this work looks more broadly at issues resulting from regular expressions (including ReDoS issues, which were also present in two PRs in our dataset, Section 4.1.3).

Prior work has surveyed developers to identify pain points associated with regular expression usage (Chapman and Stolee 2016; Michael et al. 2019). Rather than surveying developers, this work explored the discussions in regex-related PRs. Pain points were revealed indirectly through the fix patterns (e.g., issues with escaping literals are common and likely a pain point), and bug characteristics.

Regarding the prior work on regular expression tools, Rex (Veanes et al. 2010) explores regular expression constraints and generates its matching strings, `dk.brics.automaton` (Møller 2017) transforms a regex into an automaton, EGRET (Larson and Kirk 2016) generates both a set of accepted strings and a set of rejected strings for the given regex, and Spishak et al. (2012) provides a type system to validate regular expression syntax at compile time. Regex-opt (Perl-compatible regular expression optimizer 2021) and regexp-tree (regexp-tree - npm 2021) are regex optimizers that make regexes easier to read and reduce the risks of ReDoS. Complementary to these works, this paper shows that incorrect regex semantics and compile errors are practical problems that developers need help with. We also show that regex bugs are bigger than rejecting valid strings since data evolution and changing requirements contribute to incorrect regex semantics. Beyond regex semantics, developers are also concerned with bad smells in regular expressions.

**Software Bugs and Classification** GitHub has become a popular hosting site for organizations large and small to make their projects available to their teams and the public. Pull requests are created when a developer wants their changes to be integrated into a project; sometimes these are linked to a GitHub issue or another bug reporting software. Pull requests are reviewed and discussed before being merged.

The lens through which researchers study bugs is typically a bug report (Tan et al. 2014; Di Franco et al. 2017; Zhong and Su 2015; Ma et al. 2017; Zhang et al. 2018). GitHub pull requests (Gousios and Zaidman 2014; Majumder et al. 2019; Gousios et al. 2014; Rahman and Roy 2014) provide a different lens as they contain a proposed (or actual, in the case of a merged PR) change.

Similar research to ours is bug classification (Maalej and Nabil 2015; Herzig et al. 2013; Ohira et al. 2016). Some research targets emerging applications, such as TensorFlow bugs (Zhang et al. 2018) and Blockchain bugs (Wan et al. 2017), while others target distributed systems such as node change bugs (Lu et al. 2019) and concurrency bugs (Lu et al.

2008). More specific bug types include bugs in exception-related code (Coelho et al. 2015), bugs in patches (Yin et al. 2011), bugs in test code (Vahabzadeh et al. 2015), numerical bugs (Di Franco et al. 2017), string-related bugs (Eghbali and Pradel 2020), performance bugs (Selakovic and Pradel 2016), and cross-project correlated bugs (Ma et al. 2017). Our study joins this list with its focus on regex-related bugs. It is interesting to see that 37% of the string-related bugs (Eghbali and Pradel 2020) are caused by regular expressions with six recurring bug patterns. Our study of regex-related bugs and the string-related bug study are complementary to each other.

Bugs are often categorized in terms of root causes and manifestations (Lu et al. 2008; Di Franco et al. 2017; Zhang et al. 2018; Selakovic and Pradel 2016; Yin et al. 2011), bug patterns (Lu et al. 2008; Lou et al. 2020), and fix strategies (Lu et al. 2008; Ma et al. 2017; Selakovic and Pradel 2016). Tan et al. (2014) conduct a temporal analysis to study the trend of different types of bugs with software evolution. Zhong and Su (2015) evaluate the differences between bug fixes by programmers and the fixes by automatic program repair. Selakovic and Pradel (2016) measure the complexity of optimization code changes. Wan et al. (2017) evaluate the relationship between bug type and bug fixing time. We adopt the approach of using root causes and manifestations to describe regex-related bugs and the approach of using fix strategies to describe bug resolution.

In addition to bug studies, there is also lots of research focused on code refactoring to categorize or detect code smells and design smells in source code (Palomba et al. 2013; Moha et al. 2009; Sharma et al. 2016) and to understand the mutual impact between those bad smells and the software development process (Khomh et al. 2009; Tufano et al. 2015). As many of the PRs were addressing code smells and design smells, our work is related to this literature as well.

**API and Library Usage** Our study also have overlaps with research on library and API usages. The library usage studies have explored the motivation of library migration (Kabinna et al. 2016), how to choose the best library candidate (Mileva et al. 2009), which version should be used (Mileva et al. 2009), and how to automate the migration of the underlying API library from one to another (Wu et al. 2015; Kapur et al. 2010; Teyton et al. 2012). The API usage studies have explored mining API usage patterns (Zhong et al. 2009), detecting API misuses (Gu et al. 2019; Amann et al. 2018; Kechagia et al. 2019; Amann et al. 2016; Bae et al. 2014), studying API evolution (Zhang et al. 2021; Shi et al. 2011; Dig and Johnson 2006; Kim et al. 2011), and automatic refactoring for API changes (Perkins 2005; Thung et al. 2020) via documentation (Shi et al. 2011; Ko et al. 2014; Zhang et al. 2021; Dig and Johnson 2006), source code and its change history (Thung et al. 2020; Dig and Johnson 2006), and bytecode (Perkins 2005). The majority of API-misuse detectors are via static analysis and suffer from low precision and recall in practice (Gu et al. 2019; Amann et al. 2018) while Catcher (Kechagia et al. 2019) is a hybrid detector for crash-prone API misuses.

In this work we include both static and dynamic API changes since the missing of API parameter validation, redundant API calls, and missing or incorrect exception handling are all regard as API misuses (Gu et al. 2019). We do not distinguish the usage of library from that of API since they are connected closely and one can be studied for the other. For example, Mileva et al. use API popularity to recommend which library to use (Mileva et al. 2009) or to identify and predict API usage trends over time (Mileva et al. 2010).

**Test Code Studies** Two popular test code study areas are test code evolution and testing behavior. The test code evolution can be on either macro-level or the micro-level categorizing why and how the test code evolves. Zaidman et al. (2008) and Marsavina et al. (2014) study the testing strategies and the co-evolution patterns between production code and test code evolution. Pinto et al. (2012) calculate the distribution of added, removed, and modified test cases and investigate the reasons for different test case changes. While our study of test code is also micro-level and focus on test code addition, deletion, and modification, we focus only on regex-related code changes.

Testing behavior research has many facets. Gousios et al. (2014) report that 33% of pull requests include test code modifications in 291 selected Ruby, Python, Java and Scala projects. It also finds that the inclusion of test code does not affect the time or the decision to merge a pull request. Kochhar et al. (2013) study the distribution of test cases across 50,000 GitHub projects. Pham et al. (2013) is a study of testing behaviors on GitHub and points out that developers' demand for test code in pull requests is influenced by the size of code changes, the types of contributions, and the estimated effort. In our work, we utilize the test case detection conventions listed in prior work (Gousios and Zaidman 2014) and manually find the test case changes. Test code changes are also an important factor in constructing real-world bug benchmarks. Since the availability of benchmarks facilitates software testing, debugging, and automated repairing techniques, changed test cases are identified to guarantee the reproducibility of bugs (Widyasari et al. 2020; Just et al. 2014; Gyimesi et al. 2019; Madeiral et al. 2019; Saha et al. 2018).

## 10 Conclusion

The regular expression is one of the primary culprits of string-related bugs. While it is acknowledged that regular expressions are difficult to use correctly and poorly tested, there is little knowledge about what regular expression problems could happen in real-world software code and the consequences of those problems. Prior work exploring why regular expression testing is difficult is also scarce.

This paper presents a comprehensive study of 356 regex-related bugs from 350 merged pull requests in Apache, Mozilla, Facebook, and Google GitHub repositories where the regular expression problems are studied carefully via bug classification, bug fix, and the test code in the bug fix. The results of this study include: 1) a spectrum of regular expression root causes and manifestations with their frequency in real-world software; 2) ten common patterns of regex bug fixes; and 3) quantitative and qualitative analysis of pull request test code changes. We demonstrate that regular expression problems are not trivial, as regex-related PRs take more time and more lines of code to fix compared to general pull requests. Our study shows that regular expression bugs are not independent of the source code, but are influenced by the software evolution and the code quality. The analysis on test code changes indicates that some difficulty in testing regexes lies in the location of the regex and that regex is seldom to be tested alone. Our results and findings provide an overview of regular expression bugs and motivates future work on techniques and tools to solve practical regular expression problems.

**Acknowledgements** This material is based upon work supported by the National Science Foundation under Grant No. 1714699.

## References

Amann S, Nadi S, Nguyen HA, Nguyen TN, Mezini M (2016) Mubench: A benchmark for api-misuse detectors. In: Proceedings of the 13th International Conference on Mining Software Repositories. MSR '16, vol 4. Association for Computing Machinery, New York, NY, USA, pp 464–467. https://doi.org/10.1145/2901739.2903506

Amann S, Nguyen HA, Nadi S, Nguyen TN, Mezini M (2018) A systematic evaluation of static api-misuse detectors. IEEE Trans Softw Eng 45(12):1170–1188

Apache software foundation (2020) https://github.com/apache

Bae S, Cho H, Lim I, Ryu S (2014) Safewapi: Web api misuse detector for web applications. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. pp 507–517

Bai GR, Clee B, Shrestha N, Chapman C, Wright C, Stolee KT (2019) Exploring tools and strategies used during regular expression composition tasks. In: Proceedings of the 27th international conference on program comprehension. IEEE Press, pp 197–208

Bartoli A, De Lorenzo A, Medvet E, Tarlao F (2016) Inference of regular expressions for text extraction from examples. IEEE Trans Knowl Data Eng 28(5):1217–1230. https://doi.org/10.1109/TKDE.2016.2515587

Beller M, Gousios G, Zaidman A (2017) Oops, my tests broke the build: An explorative analysis of travis ci with github. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR). IEEE, pp 356–367

Brown WH, Malveau RC, McCormick HW, Mowbray TJ (1998) AntiPatterns: refactoring software, architectures, and projects in crisis

Chapman C, Stolee KT (2016) Exploring regular expression usage and context in python. In: Proceedings of the 25th international symposium on software testing and analysis. ACM, pp 282–293

Chapman C, Wang P, Stolee KT (2017) Exploring regular expression comprehension. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering. IEEE Press, pp 405–416

Chen Q, Wang X, Ye X, Durrett G, Dillig I (2020) Multi-modal synthesis of regular expressions. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation. pp 487–502

Cody-Kenny B, Fenton M, Ronayne A, Considine E, McGuire T, O'Neill M (2017) A search for improved performance in regular expressions. In: Proceedings of the genetic and evolutionary computation conference. ACM, pp 1280–1287

Coelho R, Almeida L, Gousios G, van Deursen A (2015) Unveiling exception handling bug hazards in android based on github and google code issues. In: 2015 IEEE/ACM 12th working conference on mining software repositories. IEEE, pp 134–145

Davis JC, Coghlan CA, Servant F, Lee D (2018) The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In: The ACM joint European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)

Davis JC, Michael IV, Louis G, Coghlan CA, Servant F (2019) Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ACM, pp 443–454

Davis JC, Moyer D, Kazerouni AM, Lee D (2019) Testing regex generalizability and its implications: A large-scale many-language measurement study. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 427–439

Davis JC, Servant F, Lee D (2021) Using selective memoization to defeat regular expression denial of service (redos). In: 2021 IEEE symposium on security and privacy (SP)

Di Franco A, Guo H, Rubio-González C (2017) A comprehensive study of real-world numerical bug characteristics. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering. IEEE Press, pp 509–519

Dig D, Johnson R (2006) How do apis evolve? a story of refactoring. J Softw Maint Evol Res Pract 18(2):83–107

Eghbali A, Pradel M (2020) No strings attached: An empirical study of string-related software bugs. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 956–967

Facebook (2020) https://github.com/facebook

Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston

Githut - programming languages and github (2014) https://githut.info/

Google (2020) https://github.com/google

Gousios G, Pinzger M, Deursen AV (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering. ACM, pp 345–355

Gousios G, Zaidman A (2014) A dataset for pull-based development research. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 368–371

Github graphql api v4 2019 (2020) https://developer.github.com/v4/

Gu Z, Wu J, Liu J, Zhou M, Gu M (2019) An empirical study on api-misuse bugs in open-source c programs. In: 2019 IEEE 43rd annual computer software and applications conference (COMPSAC), vol 1. IEEE, pp 11–20

Gyimesi P, Vancsics B, Stocco A, Mazinanian D, Beszédes A., Ferenc R, Mesbah A (2019) Bugsjs: a benchmark of javascript bugs. In: 2019 12th IEEE conference on software testing, validation and verification (ICST). IEEE, pp 90–101

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 392–401

Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, pp. 437–440

Kabinna S, Bezemer CP, Shang W, Hassan AE (2016) Logging library migrations: A case study for the apache software foundation projects. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). IEEE, pp 154–164

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories. pp. 92–101

Kapur P, Cossette B, Walker RJ (2010) Refactoring references for library migration. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. pp. 726–738

Kechagia M, Devroey X, Panichella A, Gousios G, van Deursen A (2019) Effective and efficient api misuse detection via exception propagation and search-based testing. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis. pp 192–203

Khomh F, Di Penta M, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th working conference on reverse engineering. IEEE, pp 75–84

Kim M, Cai D, Kim S (2011) An empirical investigation into the role of api-level refactorings during software evolution. In: Proceedings of the 33rd international conference on software engineering. pp 151–160

Ko D, Ma K, Park S, Kim S, Kim D, Le Traon Y (2014) Api document quality for resolving deprecated apis. In: 2014 21st Asia-Pacific software engineering conference, vol 2. IEEE, pp 27–30

Kochhar PS, Bissyandé T. F., Lo D, Jiang L (2013) Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In: 2013 17th european conference on software maintenance and reengineering. IEEE, pp 353–356

Larson E, Kirk A (2016) Generating evil test strings for regular expressions. In: 2016 IEEE international conference on software testing, verification and validation (ICST). IEEE, pp 309–319

Locascio N, Narasimhan K, DeLeon E, Kushman N, Barzilay R (2016) Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pp 1918–1923

Lou Y, Chen Z, Cao Y, Hao D, Zhang L (2020) Understanding build issue resolution in practice: Symptoms and fix patterns. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2020. Association for Computing Machinery, New York, pp 617–628. https://doi.org/10.1145/3368089.3409760

Lu J, Chen L, Li L, Feng X (2019) Understanding node change bugs for distributed systems. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 399–410

Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. pp 329–339

Ma W, Chen L, Zhang X, Zhou Y, Xu B (2017) How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE). IEEE, pp 381–392

Maalej W, Nabil H (2015) Bug report, feature request, or simply praise? on automatically classifying app reviews. In: 2015 IEEE 23rd international requirements engineering conference (RE). IEEE, pp 116–125

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears: An extensible java bug benchmark for automatic program repair studies. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 468–478

Majumder S, Chakraborty J, Agrawal A, Menzies T (2019) Why software projects need heroes (lessons learned from 1100+ projects). arXiv:1904.09954

Mann-Whitney-Wilcoxon Test | R Tutorial (2020) http://www.r-tutor.com/elementary-statistics/non-parametric-methods/mann-whitney-wilcoxon-test

Marsavina C, Romano D, Zaidman A (2014) Studying fine-grained co-evolution patterns of production and test code. In: 2014 IEEE 14th international working conference on source code analysis and manipulation. IEEE, pp 195–204

Michael LG, Donohue J, Davis JC, Lee D, Servant F (2019) Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In: ACM international conference on automated software engineering (ASE). ACM

Mileva YM, Dallmeier V, Burger M, Zeller A (2009) Mining trends of library usage. In: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. pp 57–62

Mileva YM, Dallmeier V, Zeller A (2010) Mining api popularity. In: International academic and industrial conference on practice and research techniques. Springer, pp 173–180

Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2009) Decor: A method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36

Møller A (2017) dk.brics.automaton – finite-state automata and regular expressions for Java. http://www.brics.dk/automaton/

Mozilla (2020) https://github.com/mozilla

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empir Softw Eng 22(6):3219–3253

Ohira M, Yoshiyuki H, Yamatani Y (2016) A case study on the misclassification of software performance issues in an issue tracking system. In: 2016 IEEE/ACIS 15th international conference on computer and information science (ICIS). IEEE, pp 1–6

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: Proceedings of the 28th IEEE/ACM international conference on automated software engineering. IEEE Press, pp 268–278

Park JU, Ko SK, Cognetta M, Han YS (2019) Softregex: Generating regex from natural language descriptions using softened regex equivalence. In: Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP). pp 6426–6432

Pearson's moment coefficient of skewness | a blog on probability and statistics (2015) https://probabilityandstats.wordpress.com/tag/pearsons-moment-coefficient-of-skewness/

Perkins JH (2005) Automatically generating refactorings to support api evolution. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering. pp 111–114

Pham R, Singer L, Liskin O, Figueira Filho F, Schneider K (2013) Creating a shared understanding of testing culture on a social coding site. In: 2013 35th international conference on software engineering (ICSE). IEEE, pp 112–121

Pingclasai N, Hata H, Matsumoto KI (2013) Classifying bug reports to bugs and other requests using topic modeling. In: 2013 20Th asia-pacific software engineering conference (APSEC), vol 2. IEEE, pp 13–18

Pinto LS, Sinha S, Orso A (2012) Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, pp. 1–11

Pygithub - pygithub 1.45 documentation (2020) https://pygithub.readthedocs.io/en/latest/

Regex use vs. regex abuse (2005) https://blog.codinghorror.com/regex-use-vs-regex-abuse

Stack overflow (2021) c# - \ d less efficient than [0-9] - stack overflow. https://stackoverflow.com/questions/16621738/d-less-efficient-than-0-9

Perl-compatible regular expression optimizer (2021) https://bisqwit.iki.fi/source/regexopt.html

regexp-tree - npm (2021) https://www.npmjs.com/package/regexp-tree

Replacing a complex regular expression with a simple parser (2017) https://www.honeybadger.io/blog/replacing-regular-expressions-with-parsers/

Rahman MM, Roy CK (2014) An insight into the pull requests of github. In: Proc. MSR, vol 14

Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR (2018) Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 15th international conference on mining software repositories. pp 10–13

Selakovic M, Pradel M (2016) Performance issues and optimizations in javascript: an empirical study. In: Proceedings of the 38th international conference on software engineering. ACM, pp 61–72

Sharma T, Fragkoulis M, Spinellis D (2016) Does your configuration code smell? In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). IEEE, pp 189–200

Shen Y, Jiang Y, Xu C, Yu P, Ma X, Lu J (2018) Rescue: Crafting regular expression dos attacks. In: 2018 33rd IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 225–235

Shi L, Zhong H, Xie T, Li M (2011) An empirical study on evolution of api documentation. In: International conference on fundamental approaches to software engineering. Springer, pp 416–431

Skewness | R Tutorial (2021) http://www.r-tutor.com/elementary-statistics/numerical-measures/skewness

Spishak E, Dietl W, Ernst MD (2012) A type system for regular expressions. In: Proceedings of the 14th workshop on formal techniques for java-like programs. ACM, pp 20–26

Staicu CA, Pradel M (2018) Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp 361–376

Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empir Softw Eng 19(6):1665–1705

Teyton C, Falleri JR, Blanc X (2012) Mining library migration graphs. In: 2012 19th working conference on reverse engineering. IEEE, pp 289–298

Thung F, Haryono SA, Serrano L, Muller G, Lawall J, Lo D, Jiang L (2020) Automated deprecated-api usage update for android apps: How far are we? In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 602–611

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the 37th international conference on software engineering-volume 1. IEEE Press, pp. 403–414

Vahabzadeh A, Fard AM, Mesbah A (2015) An empirical study of bugs in test code. In: 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 101–110

Veanes M, De Halleux P, Tillmann N (2010) Rex: Symbolic regular expression explorer. In: 2010 Third international conference on software testing, verification and validation. IEEE, pp 498–507

Wan Z, Lo D, Xia X, Cai L (2017) Bug characteristics in blockchain systems: a large-scale empirical study. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR). IEEE, pp 413–424

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. pp 103–113

Wang P, Gina R, Stolee KT (2019) Exploring regular expression evolution. In: 2019 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 502–513

Wang P, Stolee KT (2018) How well are regular expressions tested in the wild? In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ACM, pp 668–678

Wang P, Brown C, Jennings JA, Stolee KT (2020) Dataset for Comprehensive Study of Regular Expression Bugs. https://figshare.com/articles/dataset/A_Dataset_for_Comprehensive_Study_of_Regular_Expression_Bugs/13234538, hhttps://doi.org/10.6084/m9.figshare.13234538.v1

Wang X, Hong Y, Chang H, Park K, Langdale G, Hu J, Zhu H (2019) Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association, Boston, MA, pp 631–648

When you should NOT use Regular Expressions? (2011) https://softwareengineering.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions

Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y et al (2020) Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. pp 1556–1560

Wu L, Wu Q, Liang G, Wang Q, Jin Z (2015) Transforming code with compositional mappings for api-library switching. In: 2015 IEEE 39th annual computer software and applications conference, vol. 2. IEEE, pp 316–325

Wüstholz V., Olivo O, Heule MJ, Dillig I (2017) Static detection of dos vulnerabilities in programs that use regular expressions. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp 3–20

Ye X, Chen Q, Wang X, Dillig I, Durrett G (2020) Sketch-driven regular expression generation from natural language and examples. Transactions of the Association for Computational Linguistics 8:679–694

Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L (2011) How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. ACM, pp 26–36

Yu S, Xu L, Zhang Y, Wu J, Liao Z, Li Y (2018) Nbsl: A supervised classification model of pull request in github. In: 2018 IEEE international conference on communications (ICC). IEEE, pp 1–6

Zaidman A, Van Rompaey B, Demeyer S, Van Deursen A (2008) Mining software repositories to study co-evolution of production & test code. In: 2008 1st international conference on software testing, verification, and validation. IEEE, pp 220–229

Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. ACM, pp 129–140

Zhang Z, Yang Y, Xia X, Lo D, Ren X, Grundy J (2021) Unveiling the mystery of api evolution in deep learning frameworks a case study of tensorflow 2. In: 2021 IEEE/ACM 43rd international conference on software engineering: software engineering in practice (ICSE-SEIP). IEEE, pp. 238–247

Zhong H, Su Z (2015) An empirical study on real bug fixes. In: Proceedings of the 37th international conference on software engineering-volume 1. IEEE Press, pp 913–923

Zhong H, Xie T, Zhang L, Pei J, Mei H (2009) Mapo: Mining and recommending api usage patterns. In: European conference on object-oriented programming. Springer, pp 318–343

Zhong Z, Guo J, Yang W, Peng J, Xie T, Lou JG, Liu T, Zhang D (2018) Semregex: A semantics-based approach for generating regular expressions from natural language specifications. In: Proceedings of the 2018 conference on empirical methods in natural language processing

## Affiliations

**Peipei Wang**[1] · **Chris Brown**[2] · **Jamie A. Jennings**[3] · **Kathryn T. Stolee**[3]

Chris Brown
dcbrown@vt.edu

Jamie A. Jennings
jjennings@ncsu.edu

Kathryn T. Stolee
ktstolee@ncsu.edu

1 Infrastructure System Lab, ByteDance, WA, USA

2 Department of Computer Science, Virginia Tech, VA, USA

3 Department of Computer Science, North Carolina State University, NC, USA