An Empirical Study on Regular Expression Bugs

Peipei Wang
North Carolina State University
Raleigh, NC, USA
pwang7@ncsu.edu

Chris Brown
North Carolina State University
Raleigh, NC, USA
dcbrow10@ncsu.edu

Jamie A. Jennings
North Carolina State University
Raleigh, NC, USA
jjennings@ncsu.edu

Kathryn T. Stolee
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

ABSTRACT

Understanding the nature of regular expression (regex) issues is important to tackle practical issues developers face in regular expression usage. Knowledge about the nature and frequency of various types of regular expression issues, such as those related to performance, API misuse, and code smells, can guide testing, inform documentation writers, and motivate refactoring efforts. However, beyond ReDoS (Regular expression Denial of Service), little is known about to what extent regular expression issues affect software development and how these issues are addressed in practice.

This paper presents a comprehensive empirical study of 350 merged regex-related pull requests from Apache, Mozilla, Facebook, and Google GitHub repositories. Through classifying the root causes and manifestations of those bugs, we show that incorrect regular expression behavior 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%). By studying the code changes of regex-related pull requests, we observe that 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. The results of this study contribute to a broader understanding of the practical problems faced by developers when using regular expressions.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software defect analysis**.

KEYWORDS

Regular expression bug characteristics, pull requests, bug fixes

ACM Reference Format:
Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. 2020. An Empirical Study on Regular Expression Bugs. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387464

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387464

1 INTRODUCTION

Regular expression research in software engineering has explored performance issues [18], comprehension [17], translation between languages [21, 22], and test coverage [46]. These efforts are motivated by the assumption 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. Yet, the nature of these issues related to regular expressions, aside from ReDoS vulnerabilities [20], is largely unknown.

Knowledge about the frequency of various types of regular expression issues, such as those related to performance, API misuse, and code smells, can guide testing, inform documentation writers, and motivate refactoring efforts. 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.

As a lens into issues developers face and fix, we explore merged pull requests (PRs) related to regular expressions (*regex-related pull requests*). The assumption is that pull requests that are merged represent issues in code that developers find worthy of fixing. 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 in multiple programming languages. Prior work suggests that there are significant differences in some regex characteristics across programming languages [22], and our findings echo this: we likewise find differences in bug characteristics across languages.

Our main findings are a classification of the regular expression bugs addressed by developers. For example, 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. The contributions of this work are:

- The first comprehensive empirical study on regular expression bugs in real-world open-source projects.
- Identification of root causes and manifestations of regular expression bugs with 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.

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 [25]: 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).

3 STUDY

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

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 expression issues. 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 issues 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 [7], Mozilla [13], Google [11], and Facebook [9]. Using the GitHub GraphQL API [10], 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 [4]. This resulted in 664 merged pull requests from 195 GitHub repositories in the 4 organizations.

Figure 1: Example of Regex Addition from a pull request in JavaScript (mozilla/zamboni#1442)

```
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)));
```

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, Figure 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.

3.1.3 Final Dataset Description. The final dataset of 350 regex-related PRs comes from 135 GitHub repositories. Of these, 86 are from Apache repositories, 162 are from Mozilla repositories, 66 are from Facebook repositories, and 36 are from Google repositories. When analyzing regex-related code changes, we considered the overall code differences before and after the PR, hence avoiding issues from reworked commits (Peril VII [27]). 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 *pull request bugs*. Our final data are available [8].

3.2 RQ1 Analysis: Bug Characteristics

With the 356 pull request 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). For example, 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 smells*. PR mozilla/addons-server#10352 addresses a problem is described in a GitHub issue, which identifies an error caused by the incorrect flag in *regex API* with the manifestation of *exception handling* [2]. In a JIRA bug report related to PR apache/ambari#760, the problem being addressed is *incorrect regex behavior* because valid URLs are rejected and the scope of the regular expression needs to be expanded.

[1] While we avoid many perils of mining GitHub [27] 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 7.
[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.

After card sorting is complete, eight manifestations of three root causes of regex-related bugs are identified. Four of the eight manifestations are further broken into categories and sub-categories according to the common characteristics shared by the bugs. The hierarchy of the 356 pull request bugs is presented in Table 1.

3.3 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.

3.3.1 Complexity of Regex-related PR Fixes. To understand if regex-related bugs are similar in complexity to other software bugs, we compare our regex-related PRs (*regexPRs*) with a public dataset of PRs from GitHub projects that use PRs in their development cycle [25] (*allPRs*). We selected four features from the prior work that represent the complexity of the fix or the complexity of reviewing the PR. For the complexity of reviewing the PR, we chose the number of minutes from PR initialization to merge (*mergetime_minutes*). For measuring the complexity of the fix in the PR, 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* and *test_churn*. 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 from the testing code.

The metrics for bug fix complexity in our dataset (*regexPRs*) are obtained through the PyGithub [14] library, which provides APIs to retrieve GitHub resources. The *allPRs* dataset [25] 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. We used the Mann-Whitney-Wilcoxon Test [12] to investigate whether our dataset, *regexPRs*, and the *allPRs* dataset have the same distribution. These comparison results are presented in Table 2.

3.3.2 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 being 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 find it is parsed with regular expression syntax and a regular expression instance or object is 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 Figure 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 Figure 1, there is no regex string prior to the PR whereas line 4 introduces regular expression `/\d+/`.

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 [45].

Figure 2: Example of Regex API Changes from a pull request in Java (google/ExoPlayer#3185)

```
1   currentLine = subripData.readLine();
2 - Matcher matcher = SUBRIP_TIMING_LINE.matcher(currentLine);
3 - if (matcher.matches()) {
4 + Matcher matcher = currentLine == null ? null :
5         SUBRIP_TIMING_LINE.matcher(currentLine);
6 + if (matcher != null && matcher.matches()) {
```

Similar to regex addition, a regular expression removal (R_{rm}) is counted when the code before a PR contains more regexes than after the PR. 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).

The regular expression API change (R_{API}) encapsulates changes to the APIs being used statically and dynamically. This includes modifying the method itself on a certain call site and reducing the execution frequency of that call site. For modifying the API method, we counted only when the regex object shows up both before and after the PR. Therefore, API methods introduced with R_{add} or removed with R_{rm} are excluded. Take Figure 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 could also be about how the API methods are executed in run-time. For example, constructing regular expression objects statically rather than on-the-fly. The PR in Figure 2 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).

3.3.3 Recurring Patterns for Fixing Regular Expression Bugs. To find the common fix patterns, we manually examined the code changes in pull requests caused by either regex or API. Since we are more interested in fixing regular expression bugs, the regex-related PRs caused by other code are out of the scope of common fix patterns of regex bugs. Each regex-related change is regarded as a different pattern, and similar changes are grouped together. We chose ten recurring patterns to represent fix strategies for common regular expression problems.

4 RQ1: BUG CATEGORIES

As is done in prior work on categorizing software bugs, we identified the *root cause* and *manifestation* of the bugs [23, 30, 39, 42, 48]. 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 pull request bugs related to regular expressions, 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 1. When the root cause is the *regex*, the regex itself caused an issue; examples include incorrect behavior, a compile error, or a code smell. When

Table 1: The hierarchy for the 356 pull request bugs including root causes, manifestation, categories, and sub-categories.

Root Cause	Manifestation	Category (Sub-Category)		Count (%) in (sub)Category	Count (%) in Manifestation	Count (%) in Root Cause
Regex	Incorrect Behavior	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%)

the *regex api* is the root cause, this means the API was deprecated, the wrong flags were 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 was modified in a fix, but are not the root cause of the issue).

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 1). 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 1). 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 behaviorally equivalent task, making the manifestation a bad smell rather than a fault. 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 behavior*, *compile error*, and *bad smells*.

4.1.1 Regex: Incorrect Behavior. *Incorrect Behavior* is the dominant manifestation for bugs with the regex as the root cause (75.7%, 165/218). Table 1 shows the four categories of this manifestation: rejecting valid string, accepting invalid strings, both rejecting valid and accepting invalid strings, and incorrect extraction. Rejecting valid strings represents 61.8% of the incorrect behavior bugs. This reinforces the observation that developers prefer to compose a conservative regex to an overly liberal one [34] and tend to expand the scope of regular expressions as software evolves [45].

Two primary factors seem to contribute to incorrect regex behavior. 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 behavior may become obsolete (e.g., PR apache/cordova-ios#376). 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 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 1: *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 [20] vulnerability[3]). *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" [4]. 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 the incorrect behavior, the escape characters in this sub-category do not cause a behavioral issue. *Unused/duplicated regex* refers to regexes in code that are no longer needed (7/8) or that are duplicated (1/8), with the former being more common.

Summary: Most incorrect regular expression behavior occurs 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 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. This is a unique feature in JavaScript *stateful* regex methods (i.e., `RegExp.test` and `RegExp.exec`). Besides the stateful methods, other incorrect API usage leading to incorrect computation includes passing arguments into the wrong method, failing to process multiline 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 1 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 code is behaving properly except for the 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 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 [5]. Similarly, when flags argument is no longer supported [6] in JavaScript regex APIs, `input.replace('<', '<', 'g')` has to be changed into `replace(/</g, '<')` (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 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 pull request 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

[3]Since ReDoS cares about the time complexity of running the regular expression, we regard it as performance issue.
[4]https://eslint.org/docs/2.0.0/rules/no-regex-spaces

[5]https://jakarta.apache.org/oro/
[6]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace

Table 2: Comparing selected features of regex-related PRs (*regexPRs*) to merged PRs (*allPRs*) from prior work [25].

Feature	Meaning	Dataset	5%	mean	median	95%	skewness	p-value
mergetime_minutes	Minutes from PR initialization to merge	*allPRs*	0.00	10,529.07	405.00	43,685.05	10.99	-
		regexPRs	11.93	10,212.00	1,307.46	47,589.74	6.73	8.139e-13***
num_commits	Number of commits in the PR	*allPRs*	1.00	3.94	1.00	11.00	16.75	-
		regexPRs	1.00	2.67	1.00	8.00	7.97	0.3635
code_churn	Modified lines of code in the PR	*allPRs*	0.00	324.15	15.00	1,047.00	32.44	-
		regexPRs	2.00	615.72	27.00	786.15	18.01	1.075e-08***
files_changed	number of files changed in the PR	*allPRs*	1.00	11.84	2.00	30.00	93.62	-
		regexPRs	1.00	6.78	2.00	23.65	8.20	0.3068

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

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/react6804).

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 (apache/accumulo#628), which means the root cause is unsanitized log messages, and sanitizing them is a new feature. Table 1 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: Regexes are involved in PRs even when the regex or its APIs are not the root cause.

5 RQ2: 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

We hypothesize that regex-related PRs differ from most other PRs. We evaluate this hypothesis by comparing characteristics of regex-related PRs to PRs from a public dataset of representative PRs from GitHub projects that use PRs in their development cycle [25]. Table 2 shows the pull request feature distributions for our dataset (*regexPRs*) and the merged PRs from prior work (*allPRs*), as described in Section 3.3.1. We compare the distributions of each feature across the datasets using a Mann-Whitney-Wilcoxen test 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 [1, 5]. 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 2, *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 longer to merge (*mergetime_minutes*) and involve more lines of code (*code_churn*), and these differences are significant at $\alpha = 0.001$. Our conclusion is that regex-related PRs are different than general PRs.

Summary: The fixes in regex-related PRs are significantly different from general PRs. 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: a regex addition (R_{add}), edit (R_{edit}), or removal (R_{rm}), or a regex API is modified (R_{API}). Table 3 presents the distribution of regex changes over the 356 PR bugs with noted dominant type of regex changes.

Table 3: Distribution of the four types of regex-related changes over different root causes and manifestations. 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.

Root Cause	Manifestation (Category)		#PR	R_{add}	R_{edit}	R_{rm}	R_{API}
Regex	Incorrect Behavior		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)

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 changes 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.

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 1), 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 behavior could be fixed by replacing the regex with an existing parser (See Pattern 4 in Table 4). When incorrect regex behavior 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 4). When the incorrect regex behavior 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, 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 4 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 behavior 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).

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.

Table 4: Recurring patterns to fix regular expression bugs. 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	Description	#PR	Example Before/After
1	Correctly escaping regex literals	17	Before: `regex="a.png"` After: `regex="a\.png"`
2	Extend or shrink the character class	17	Before: `value_regex = r'[_\w]+'` After: `value_regex = r'[_\-\w]+'`
3	Replace regex with string methods	15	Before: `if re.match(".*error.*",message):` After: `if "error" in message:`

4	Replace regex with existing parser	11	Before: `EMAIL_REGEX_PATTERN.matcher(email).matches();`
			After: `import javax.mail.internet.InternetAddress;`
			`InternetAddress emailAddr = new InternetAddress(email);`
			`emailAddr.validate();`

5	Add or remove a regex alternation	10	Before: `regex="win32|windows"` After: `regex="wind32|windows|win64"`

6	Add or remove a regex to the regex list	9	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	Correct the type of regex representation; Language = {Python}	6	Before: `'pattern': '\d{1,2}/\d{1,2}'` After: `'pattern': r'\d{1,2}/\d{1,2}'`
8	Checking null values for regex execution	5	Before: `Matcher matcher = regex.matcher(currentLine);` After: `Matcher matcher = currentLine == null ? null :` `↪ regex.matcher(currentLine);`

9	Regex static compilation	4	Before: `String BLACKLIST = "...";`
			`boolean method(String name) {`
			`return !(name.matches(BLACKLIST));`
			`}`
			After: `Pattern BLACKLIST = Pattern.compile("...");`
			`boolean methodE(String name) {`
			`return !(BLACKLIST.matcher(name).matches());`
			`}`

10	Conditional checking before regex execution	4	Before: `Matcher m=Pattern.compile(regex).matcher(currentLine);`
			After: `if currentLine.contains("error"){`
			`Matcher m=Pattern.compile(regex).matcher(currentLine);`
			`}`

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. More patterns could be observed, but those presented in Table 4 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 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 [22] and porting regular expressions causes semantic and performance differences [21]. 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 behavior caused by stateful methods occurs 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 4).

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 [2, 3, 6] 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 [34]. 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 [16], cannot determine if a regex is compiled too often (Pattern 9, Table 4), if a string library would be more appropriate (Pattern 3, Table 4), or if a meta-character should be escaped (Pattern 1, Table 4). 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 regex and ReDoS [20] 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 4) 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). Automated performance support would help developers identify these inefficiencies sooner.

Testing Regexes. Prior work on regex testing [46] shows that only 17% regular expressions are tested. The PRs reveal that test code is not typically committed with regex changes; over 50% of PRs do not include test code edits. Providing test cases provides clarity on the intended behavior of the regex and may reduce discussions about what purpose a regex should serve. Among the 165 PRs causing incorrect regex behaviors, 47.9% (79) contain regex testing code for the regex and 49.7% (82) do not. The other four are not feasible to test because the regular expression is in either the configuration files or the testing framework itself specifying which tests to run (e.g.,mozilla/amo-validator#320).

We note that regex testing statistics from prior work [46] may be artificially low due to feasibility issues. Not all regexes can be tested in context. Regular expressions written in configuration files, for example, make testing more challenging (e.g., mozilla/amo-validator#520). 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.

7 THREATS TO VALIDITY

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

Our analysis considers only the code changes present in merged pull requests. Thus, and changes that proceed or follow the PR but are not linked to the PR were not analyzed.

Construct Validity. We analyzed 356 merged PR bugs from 4 organizations, which may not be representative of all regex-related PRs. 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.

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.

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.

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 at a specific date and time. Results may not generalize to PRs sampled from a different period.

We used GitHub's merge status in selecting PRs, which poses a threat to validity [27]. This threat is that 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. Further study is needed to assess the impact of this threat.

Among the 16 PR features [25], we only select four of them to evaluate RQ2. The comparison between *regexPRs* and the *allPRs* dataset may not hold on the other features.

8 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., [16, 17, 20–22, 45, 46]). Previous research focuses on regex feature usage in one language [16] and later on comparing regex characteristics across languages [21, 22], with a specific focus on portability issues [21]. Previous research also explores regex characteristics (e.g., size, features) at a moment in time [16, 21], or, more similar to this work, on changes to the characteristics over time through the lens of commit history [45]. Another dimension is context: some regular expression studies extract regexes from their context for analysis (e.g., [17, 45]), but others consider the execution environment (e.g., to measure test coverage [46] or identify actual ReDoS issues [20]). In this work, we analyzed regexes in multiple languages using the context from the PR, which is not available through commit history alone, in addition to properties of the regex itself.

Regex comprehension has been studied using controlled experiments [17] and composition strategies have been studied using observational studies of developers [15]. 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 [20, 41]. While the prior work [20] 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 [16, 34]. 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.

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 [23, 31, 42, 48, 49]. GitHub pull requests [24, 25, 33, 38] 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 [26, 32, 36]. Some research targets emerging applications, such as TensorFlow bugs [48] and Blockchain bugs [44], while others target distributed systems such as node change bugs [29] and concurrency bugs [30]. More specific bug types include bugs in exception-related code [19], bugs in patches [47], numerical bugs [23], performance bugs [39], and cross-project correlated bugs [31]. Our study joins this list with its focus on regex-related bugs.

Bugs are often categorized in terms of root causes and manifestations [23, 30, 39, 47, 48], bug patterns [30], and fix strategies [30, 31, 39]. Tan and Liu et al. [42] conduct a temporal analysis to study the trend of different types of bugs with software evolution. Zhong et al. [49] evaluate the differences between bug fixes by programmers and the fixes by automatic program repair. Selakovic et al. [39] measure the complexity of optimization code changes. Wan et al. [44] 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 [35, 37, 40] and to understand the mutual impact between those bad smells and the software development process [28, 43]. As many of the PRs were addressing code smells, our work is related to this literature as well.

9 CONCLUSION

Most empirical studies on regular expression bugs are focused on detecting or fixing ReDoS. The studied regular expressions are often extracted from source code, and thus real, but the whole empirical study is often separate from the environment where the regexes are executed. There is little knowledge about what regular expression problems could happen in real-world software code and the consequences of those problems.

This paper presents a study of 350 merged regular expression related pull requests from Apache, Mozilla, Facebook and Google GitHub repositories where the regular expression problems are studied carefully by bug descriptions and the source code. Our results provide not only the dominant regular expression problems but also a spectrum of regular expression root causes and manifestations. Our study shows that regular expression bugs are not independent of the source code it runs, but are influenced by the software evolution and the code quality. Furthermore, by analyzing the complexity of regular expression bug fixes, we demonstrate that regular expression bugs are not trivial problems as they take more time and more lines of code to fix compared to general bugs. We also provide ten common patterns of regex bug fixes. Our results and finding provides 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

[1] [n.d.]. Skewness | R Tutorial. http://www.r-tutor.com/elementary-statistics/numerical-measures/skewness.

[2] 2005. Regex use vs. Regex abuse. https://blog.codinghorror.com/regex-use-vs-regex-abuse.

[3] 2011. When you should NOT use Regular Expressions? https://softwareengineering.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions.

[4] 2014. GitHut - Programming Languages and GitHub. https://githut.info/.

[5] 2015. Pearson's moment coefficient of skewness | A Blog on Probability and Statistics. https://probabilityandstats.wordpress.com/tag/pearsons-moment-coefficient-of-skewness/.

[6] 2017. Replacing a Complex Regular Expression with a Simple Parser. https://www.honeybadger.io/blog/replacing-regular-expressions-with-parsers/.

[7] 2020. The Apache Software Foundation. https://github.com/apache.

[8] 2020. An Empirical Study on Regular Expression Bugs Dataset. https://figshare.com/s/802eb74c2e722ca5d8df. https://doi.org/10.6084/m9.figshare.11620083

[9] 2020. Facebook. https://github.com/facebook.

[10] 2020. Github GraphQL API v4 2019. https://developer.github.com/v4/.

[11] 2020. Google. https://github.com/google.

[12] 2020. Mann-Whitney-Wilcoxon Test | R Tutorial. http://www.r-tutor.com/elementary-statistics/non-parametric-methods/mann-whitney-wilcoxon-test.

[13] 2020. Mozilla. https://github.com/mozilla.

[14] 2020. PyGithub - PyGithub 1.45 documentation. https://pygithub.readthedocs.io/en/latest/.

[15] Gina R Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T Stolee. 2019. Exploring tools and strategies used during regular expression composition tasks. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 197–208.

[16] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 282–293.

[17] Carl Chapman, Peipei Wang, and Kathryn T Stolee. [n.d.]. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017). IEEE Press, 405–416.

[18] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2017. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 1280–1287.

[19] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 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, 134–145.

[20] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. [n.d.]. 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)* (2018).

[21] James C Davis, Louis G Michael IV, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 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, 443–454.

[22] James C Davis, Daniel Moyer, Ayaan M Kazerouni, and Dongyoon Lee. 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, 427–439.

[23] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 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, 509–519.

[24] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.

[25] Georgios Gousios and Andy Zaidman. 2014. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 368–371.

[26] Kim Herzig, Sascha Just, and Andreas Zeller. 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, 392–401.

[27] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.

[28] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 75–84.

[29] Jie Lu, Liu Chen, Lian Li, and Xiaobing Feng. 2019. Understanding Node Change Bugs for Distributed Systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 399–410.

[30] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 329–339.

[31] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 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, 381–392.

[32] Walid Maalej and Hadeer Nabil. 2015. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd international requirements engineering conference (RE)*. IEEE, 116–125.

[33] Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies. 2019. Why Software Projects need Heroes (Lessons Learned from 1100+ Projects). *arXiv preprint arXiv:1904.09954* (2019).

[34] Louis G Michael IV, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *ACM International Conference on Automated Software Engineering (ASE). ACM.*

[35] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.

[36] Masao Ohira, Hayato Yoshiyuki, and Yosuke Yamatani. 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, 1–6.

[37] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 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, 268–278.

[38] Mohammad Masudur Rahman Chanchal K Roy. 2014. An Insight into the Pull Requests of GitHub. In *Proc. MSR*, Vol. 14.

[39] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 61–72.

[40] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 189–200.

[41] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 361–376.

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

[43] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 403–414.

[44] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 413–424.

[45] Peipei Wang, Rui Gina, and Kathryn T Stolee. [n.d.]. Exploring Regular Expression Evolution. In *Software Analysis, Evolution and Reengineering (SANER), 2019 IEEE International Conference on* (2019). IEEE, 502–513.

[46] Peipei Wang and Kathryn T Stolee. 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, 668–678.

[47] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 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, 26–36.

[48] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 129–140.

[49] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 913–923.