

# ABSTRACT

WANG, PEIPEI. Analyses of Regular Expression Usage in Software Development. (Under the direction of Dr. Kathryn T. Stolee).

Although a powerful tool, a regular expression (regex) can be very complex and error prone. Regex-related faults are common whereas developers test regexes less often than the rest of the code. To better understand the issues surrounding regular expressions in software development, my work evaluates 1) regular expression bugs in software maintenance, 2) regular expression coverage in software testing, and 3) regular expression edits in software evolution.

To study regular expression bugs, I conduct an empirical study of 356 merged regex-related pull requests and build a taxonomy of regex bugs. While incorrect regular expression semantics are the dominant root cause of regex bugs (46.3%), there are also regex bugs caused by incorrect API usage (9.3%). Fixing regular expression bugs is a nontrivial task as it takes more time and more lines of code to fix them compared to the general pull requests. In regex-related pull requests where regex changes are involved, most (51%) of them do not contain test code changes, suggesting regexes may be under-tested.

To evaluate how thoroughly regular expressions are tested, I collect the regular expressions in 1,225 GitHub Java projects and apply graph-based test coverage metrics to the regex DFA and measure the coverage. Of the 18,000 regular expression call sites in the source code, only 17% of them are executed by the test suites, resulting in over 15,000 tested regexes (25% of tested call sites dynamically compose regexes). For those that are tested, the median number of test inputs is two. For nearly 42% of the tested regular expressions, only one test input is used. With some exceptions, tools such as Rex can be used to write test inputs with similar coverage to the developer tests.

I conduct another study to learn about the change history of regexes, focusing on the characteristics of regex edits, the syntactic and semantic difference of the edits, and the feature changes of edits. The results show that regular expressions from GitHub are not edited frequently, that most edits contain a change of 4-6 characters, and that over 50% of the edits in GitHub tend to expand the regex semantic scope.

The dissertation contributes to our knowledge of how regular expressions are used in software development, the issues that arise, and how those issues are resolved. These results may help guide future developer tools that specifically target regex problems.

© Copyright 2021 by Peipei Wang

All Rights Reserved

Analyses of Regular Expression Usage in Software Development

by  
Peipei Wang

A dissertation submitted to the Graduate Faculty of  
North Carolina State University  
in partial fulfillment of the  
requirements for the Degree of  
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2021

APPROVED BY:

---

Dr. Gregg Rothermel

---

Dr. Christopher Parnin

---

Dr. Bradley Reaves

---

Dr. Jamie A. Jennings

---

Dr. Kathryn T. Stolee  
Chair of Advisory Committee

## DEDICATION

To my grandparents in the heaven, my boyfriend, and my lovely cat.

## BIOGRAPHY

Peipei Wang received from Xi'an Jiaotong University her Bachelor's degree in Software Engineering (2010) and her Master's degree in Computer Science (2013). Afterwards, she decided to continue research and joined North Carolina State University. During her time at NC state, she worked with Dr. Xiaohui(Helen) Gu on distributed systems (2013-2017) and Dr. Kathryn T. Stolee (2017-2021) on software engineering. She gained industry experience through internships at IBM (2015 and 2018) and Facebook (2020).

Her research interests include software testing, program analysis, regular expression, distributed system, and cloud computing. Particularly, her interests are focused on program analysis techniques to improve regular expression comprehension, testing, and maintenance, and techniques of cloud system debugging and diagnosis to detect and repair system anomalies and bugs at the fine granularity.

## ACKNOWLEDGEMENTS

First of all, I would like to thank all the people who have helped me in the past years. They teach me that there is no shame in asking for help. I used to be afraid of asking for help and even ignore the offers of help. These people who offer their hands to me boost my confidence and encourage me to speak out about my problems and to help others as well. I am truly grateful.

I would like to acknowledge the committee members for their time and feedback on my dissertation, especially for the suggestions of my proposal which drive the direction of this body of work. To Dr. Jamie A. Jennings, for your valuable insights and discussions about regular expressions. It is a great pleasure to work and discuss research with you. Many many thanks to my excellent advisor Dr. Kathryn Stolee for her support and help in my research. She demonstrates to me what good research looks like and how to conduct research correctly. She is also a mentor to my spirit and my example of a successful female scholar.

Thanks to all other faculty, staff, and students in the Department of Computer Science at NC State. To Dr. Timothy Menzies, for his recommendation to the 2018 IBM internship opportunity where starts my interest in mining GitHub repositories. To George Rouskas, for his DGP services and for supporting my decision of changing advisor in the fourth year. To Dr. Helen Gu, for being my reason to come to North Carolina State University in the first place. To former NCSU graduate Garima Singh, for her advice about job searching and interviews. Also, I am grateful to the students I've met in the laboratories, including Vivek Nair, Anwasha Das, Rui Shu, Luke Deshotels, Ting Dai, George Mathew, Chris Brown, Gina Bai, Justin Middleton, Kai Presler-Marshall.

I would also like to thank my family for their support and patience in my years as a Ph.D. student. The celebration of my birthday thousands of miles away is the time every year when I want to go home most. Eight years is a quite long time and I missed many important family events. I missed my brother's and sister's weddings and missed saying goodbye to my grandmothers. My dear family, thanks for forgiving my absence where we should stay together as a family.

In each loss, there is a gain. Thanks to my boyfriend and my cat for their accompany. Looking back, I would thank the young naive girl whose curiosity and crave for knowledge get me through the tough days and bring this dissertation to the readers.

# TABLE OF CONTENTS

<b>LIST OF TABLES</b> . . . . .	<b>viii</b>
<b>LIST OF FIGURES</b> . . . . .	<b>x</b>
<b>Chapter 1 OVERVIEW</b> . . . . .	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Research Overview . . . . .	3
1.3 Contributions . . . . .	3
1.4 Outline . . . . .	4
<b>Chapter 2 Background and Related Work</b> . . . . .	<b>5</b>
2.1 Background . . . . .	5
2.2 Related Work . . . . .	7
2.2.1 Regular Expression . . . . .	7
2.2.2 Software Engineering Topics Related to This Dissertation . . . . .	10
<b>Chapter 3 Regular Expression Bugs</b> . . . . .	<b>15</b>
3.1 Introduction . . . . .	15
3.2 Research Questions . . . . .	17
3.3 Study . . . . .	18
3.3.1 Dataset . . . . .	18
3.3.2 Terms and Definitions . . . . .	19
3.3.3 RQ1 Analysis: Bug Characteristics . . . . .	20
3.3.4 RQ2 Analysis: Fix Characteristics . . . . .	22
3.3.5 RQ3 Analysis: Test Code Characteristics . . . . .	24
3.4 RQ1 Results: Bug Categories . . . . .	26
3.4.1 Bugs Caused by Regexes Themselves . . . . .	28
3.4.2 Bugs Caused by Regex APIs . . . . .	29
3.4.3 Bugs Caused by Other Code . . . . .	31
3.5 RQ2 Results: Bug Fix Characteristics in Regex-related PRs . . . . .	33
3.5.1 Complexity of Regex-related PR Fixes . . . . .	33
3.5.2 Changes to Regexes in PRs . . . . .	34
3.5.3 Recurring Patterns to Fix Regular Expression Bugs . . . . .	36
3.6 RQ3 Results: Test Code Characteristics in Regex-related Bugs . . . . .	38
3.6.1 Overview . . . . .	39
3.6.2 Regex-related Bugs without Test Code Changes . . . . .	40
3.6.3 Regex-related Bugs with Test Code Changes . . . . .	41
3.7 Discussion . . . . .	43
3.8 Threats to validity . . . . .	45
3.9 Summary . . . . .	47

<b>Chapter 4 Regular Expression Testing</b>	<b>48</b>
4.1 Introduction	48
4.2 Motivation	49
4.2.1 DFA with Regular Expression Matching	50
4.2.2 How Regular Are Regular Expressions?	52
4.2.3 Limitations of Code Coverage	52
4.2.4 DFA Coverage Example	53
4.3 Regular Expression Test Coverage Metrics	55
4.3.1 Graph Notation	55
4.3.2 Coverage Criteria	56
4.4 Research Questions	58
4.5 Study	59
4.5.1 Instrumentation	59
4.5.2 Coverage Analysis	61
4.5.3 Artifacts for RQ1	63
4.5.4 Artifacts for RQ2	66
4.6 Results	68
4.6.1 RQ1: Test Coverage of Regular Expressions	69
4.6.2 RQ2: Coverage with Rex	71
4.7 Threats to Validity	73
4.8 Summary	74
<b>Chapter 5 Regular Expression Evolution</b>	<b>75</b>
5.1 Introduction	75
5.2 Motivation	76
5.2.1 Regular Expression Equivalence	77
5.2.2 Regular Expression Feature Changes	77
5.3 Research Questions	78
5.4 Analysis	78
5.4.1 Syntactic Similarity	79
5.4.2 Semantic Similarity	80
5.4.3 Language Features	82
5.4.4 Implementation	82
5.5 Artifacts	84
5.5.1 GitHub Dataset	84
5.5.2 Video Dataset	85
5.6 RQ1: Regular Expression Evolution Characteristics	86
5.6.1 Edit Frequency of Regular Expressions	86
5.6.2 Runtime Errors	87
5.6.3 Regular Expression Reversions	87
5.7 RQ2: Syntactic and Semantic Evolution	88
5.7.1 Syntactic Similarity	88
5.7.2 Semantic Similarity	89

5.8	RQ3: Regular Expression Feature Changes . . . . .	94
5.8.1	Feature Vector Edits . . . . .	94
5.8.2	Feature Vector Non-Edits . . . . .	95
5.9	Discussion . . . . .	96
5.9.1	Implications . . . . .	97
5.9.2	Threats to Validity . . . . .	97
5.10	Summary . . . . .	98
<b>Chapter 6 CONCLUSION . . . . .</b>		<b>99</b>
6.1	Thesis Statement Revisited . . . . .	99
6.2	Future Work . . . . .	100
6.2.1	Beyond Structural Coverage Metrics . . . . .	101
6.2.2	String-generation Tools . . . . .	101
6.2.3	Robust Regular Expression . . . . .	102
6.2.4	Regular Expressions and Developers . . . . .	103
6.3	Epilogue . . . . .	103
<b>Bibliography . . . . .</b>		<b>104</b>

## LIST OF TABLES

Table 3.1	The number of regex-related pull requests (repositories) selected from the four GitHub organizations. . . . .	19
Table 3.2	The hierarchy for the 356 regex-related bugs including root causes, manifestation (manifest.), categories, and sub-categories (sub-category). . . . .	27
Table 3.3	Comparing selected features of regex-related PRs ( <i>regexPRs</i> ) to merged PRs ( <i>allPRs</i> ) from prior work [70]. . . . .	33
Table 3.4	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. . . . .	34
Table 3.5	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. . . . .	37
Table 3.6	The number of regex-related bugs w/o test code changes in the pull requests. . . . .	39
Table 3.7	The distribution of test code changes among the 356 regex-related bugs in each root cause and manifestation. . . . .	40
Table 3.8	The distribution of test case changes among the 171 regex-related bugs with test code changes. . . . .	41
Table 4.1	Coverage of $\backslash d+$ : $S = \{“2”, “1001”, “u”, “100u0”\}$ , $S_{succ} = \{“2”, “1001”\}$ , and $S_{fail} = \{“u”, “100u”\}$ . . . . .	56
Table 4.2	Description of 1,225 Java projects analyzed. All numbers are rounded to nearest integer except the test ratio and KLOC. . . . .	65
Table 4.3	Description of 15,096 regular expressions analyzed for RQ1. All numbers are rounded to nearest integer. . . . .	65
Table 4.4	Description of 7,926 regular expressions for RQ2. . . . .	67
Table 4.5	Description of 15,096 Regular Expressions’ test suites. All numbers are rounded to nearest integer except that success ratio rounded to two decimal places. . . . .	69
Table 4.6	Coverage values in Figure 4.7. . . . .	70
Table 4.7	Coverage values of the 7,926 regular expressions in GitHub for <i>Repo<sub>B</sub>M</i> and <i>Repo<sub>B</sub>S</i> in Figure 4.8. . . . .	72
Table 4.8	Coverage values of the 7,926 regular expressions using Rex for <i>Rex1M</i> , <i>Rex5M</i> , and <i>Rex10M</i> in Figure 4.8. . . . .	72

Table 4.9	Differences in coverage based on datasets in Figure 4.8. Hypothesis tests used paired Wilcoxon signed-rank test. Bold text identifies when one of the datasets had significantly higher coverage for all three metrics. If there was a conflict between the metrics (e.g., Set1 > Set2 for NC, and Set1 < Set2 for EPC), there was no winner. . .	73
Table 5.1	The distribution of Levenshtein distances in both GitHub and Video edits (where distance > 0). . . . .	88
Table 5.2	Edit types in GitHub and Video dataset when $k = 500$ . . . . .	90
Table 5.3	The distribution of Intersection, Addition, Removal percentages over all types of regular expression edits in GitHub and Video dataset when $k$ is 500. . . . .	90
Table 5.4	Statistics of the language features in GitHub and Video dataset ranked by the number of regular expressions in which features present (Freq), features are added, and features are removed. . . . .	95
Table 5.5	Distribution of regular expression feature changes among 256 edits in GitHub dataset and 558 edits in Video dataset. . . . .	96

## LIST OF FIGURES

Figure 3.1	Example of Regex Addition from a pull request in JavaScript (mozilla/zamboni#442). . . . .	19
Figure 3.2	Examples to illustrate identifying problems addressed in pull requests.	21
Figure 3.3	Example of Regex API Changes from a pull request in Java (google/ExoPlayer#3185). . . . .	23
Figure 4.1	Forward FullMatch DFA of regular expression ‘a+b’. . . . .	50
Figure 4.2	Backward FullMatch DFA of regular expression ‘a+b’. . . . .	51
Figure 4.3	Example to explicit the limitation of code coverage. . . . .	53
Figure 4.4	Full-match DFA for regular expression: <code>\d+\.\d+</code> . . . . .	54
Figure 4.5	Full-match DFA from RE2 [42] for the regular expression <code>\d+</code> . RE2 interprets every string as a byte stream; the range of bytes is [0-256] where [256] is added to mark the end of a string. Thus, the input string “2” would be represented as [50 256] and traverse the following path: $0 \rightarrow 1 \rightarrow 3$ . The edges marked 0-9 represent the byte range [48-57]; edges <i>not</i> 0-9 represent the byte ranges [0-47] [58-256]. . . . .	56
Figure 4.6	Visited DFA subgraphs for the regular expression ‘ <code>\d+</code> ’. For each figure, $N_0$ is the initial node 0, $N_m$ is the accept node 3, $N_e$ is the error node e. The arrows colored blue represent transitions in successful matches. The arrows colored red represent transitions in failed matches. The characters without square brackets are the literal characters in state transitions. For example, ‘u’ prompts the transition from Node 0 to Node e. [256] implies that there are no more bytes from the input string. . . . .	62
Figure 4.7	Coverage for 15,096 regular expressions. . . . .	71
Figure 4.8	Node, edge, edge-pair coverage of 7,926 regular expressions with Rex-generated ASCII inputs ( <i>Rex1M</i> , <i>Rex5M</i> , <i>Rex10M</i> ) of 7,926 regular expressions in GitHub which are used in Rex ( <i>Repo<sub>BS</sub></i> , <i>Repo<sub>BM</sub></i> ). . . . .	72
Figure 5.1	Types of semantic evolution from $r_1$ to $r_2$ . . . . .	80
Figure 5.2	Notation for sets A, B and C to compute the semantic evolution from $r_1$ to $r_2$ . . . . .	80
Figure 5.3	Example $r_1$ and $r_2$ parsed into feature vectors. LIT = Literal, DBB = Double-Bounded, QST = Questionable. . . . .	82
Figure 5.4	The distribution of Levenshtein distances for 262 GitHub edits (with distance > 0) and 647 Video edits (with distance > 0). . . . .	88
Figure 5.5	The distribution of Add, Remove, and Overlap percentages over disjoint, equal, overlap, subset and superset regular expression edits in GitHub and Video dataset when k is 500. . . . .	91

# CHAPTER

## 1

# OVERVIEW

Chapter 1 contains an *introduction* to study regular expression usage in software development, the *research overview* with research questions about regular expressions, the *contributions*, and the *outline*.

## 1.1 Introduction

A regular expression is a language describing the pattern of strings to match. Regular expressions (regexes) are used fundamentally in string searching and substitution tasks, such as data validation, classification, and extraction for various purposes (e.g., [12, 105]). More advanced uses can be seen in search engines [219], database querying [211], computational biology (e.g., DNA pattern recognition) [12], compiler lexical analysis [106], and network security [56, 81, 136].

Recent research has suggested that regular expressions are hard to understand, hard to compose, and error prone [32, 179]. This is because understanding and writing regular expressions requires both knowledge and skills. A single character mistake could cause drastically different regular expression matching behaviors. Regular expressions

are frequently used in software development and are also responsible for many errors. For example, a simple search for “regex OR regular expression” in GitHub yields 227,474 issues (and growing), with 25% of those still being open. When a regular expression is responsible for a software bug, the impact can be severe, possibly resulting in corrupted data, security vulnerabilities, or denial of service attacks) [47, 97, 174]. To make it worse, some regular expression bugs can lead to software performance issues, impacting software response time and even triggering system crashes [141, 205]. Another study on the regex development cycle [124] reveals the other difficulties of using regular expressions: hard to search for, hard to validate, and hard to document.

A plethora of theoretical research has been done on regular expression [56, 72, 210] and research on practical issues pertaining to the usage of regex is only recently (e.g., [32, 47, 124]). In fact, a survey of professional developers reveals that they test their regular expressions *less* than the rest of their code [31], and as regex bugs abound [179], this suggests that support is needed for developers.

Efforts have been put into easing the burden on developers by providing tools that make regexes easier to understand. Some tools provide debugging environments that explain string matching results and highlight the parts of regex patterns which match a certain string [140, 163]. Other tools present graphical representations (e.g., finite automata) of the regular expressions [164, 167]. Still, others either automatically generate strings according to the given regular expressions [93, 94, 129, 195] or automatically generate regular expressions according to the given list of strings [14, 110]. Specifically for regular expression testing purposes, Rex [195] is a tool for analyzing regular expressions through symbolic analysis, which we use in this work to generate strings for testing. Given a regular expression  $R$ , Rex uses the Z3 [48] SMT solver to generate members of the language by treating it as a satisfiability problem.

Still, there remains a lack of practical knowledge about bugs, bug fixes, test coverage, and evolution of regular expressions. Understanding these topics is the first step toward practical tools to support regex development. This work advances the knowledge to defend the thesis statement below:

Regular expressions in open source software have **low test coverage**, are modified **infrequently**, are overly **restrictive**, and contribute to software **bugs**.

## 1.2 Research Overview

To improve the understanding of regular expression usage in software development, my research consists of three aspects and is discussed separately: regular expression bugs, regular expression testing, and regular expression evolution.

The research begins with a *qualitative* analysis of regular expression bugs to study the causes and manifestations of regular expression bugs [200]. The fixing strategies and the test code changes along with the bug fixes are also explored to understand developers' reaction to regular expression bugs [201]. With the finding that there is not a lot of test code about regular expressions, a *quantitative* analysis of regular expression testing coverage [199] is conducted. Given the low testing coverage derived from the GitHub projects, there is a legitimate reason to expect that the under-tested regexes would bring bugs to the software. Finally, the lifespan of the regular expression is explored with the quantitative analysis through the lens of GitHub commit logs and the qualitative analysis through a fine-grained lens of programmers working in an IDE [198].

## 1.3 Contributions

In exploring how regular expressions "contribute to software bugs" (Chapter 3), this research presents:

- 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 regular expression issues.

To show that regexes "have low test coverage" (Chapter 4), this research presents:

- Application of graph-based metrics for test coverage of regular expressions: node coverage, edge coverage, and edge-pair coverage (Section 4.3).
- Test coverage evaluation of 15,096 regular expressions based on nearly 900,000 input strings from 1,225 Java projects from GitHub.
- Evaluation of test coverage achieved by the Rex symbolic analysis tool for regular expressions.

To show that regexes are "overly restricted" and are "modified infrequently" (Chapter 5) includes:

- Exploration of regular expression evolution from a coarse-grained lens of GitHub commit logs and a fine-grained lens of programmers working in an IDE.
- An empirical study of how regular expressions are edited syntactically and semantically over time, using two metrics: Levenshtein distance and semantic distance.
- An empirical study of how regular expression features changed over the process of editing regular expression.

## 1.4 Outline

The remainder of this dissertation presents the published work contributing to my research. Chapter 2 provides the background of regular expression, its usage, and related literature to this work. Chapter 3 presents the study of regular expression bugs and bug fixes [200, 201]. Chapter 4 presents the study of regular expression testing [199]. Chapter 5 provides the study of regular expression evolution with their own results and discussions [198], followed by a conclusion in Chapter 6 and a bibliography at the end of this work.

## CHAPTER

# 2

# BACKGROUND AND RELATED WORK

In Chapter 2 we present background knowledge about regex and its application in different areas, followed by the related work on regular expression and other topics related to this dissertation.

## 2.1 Background

A regular expression, or regex, is a regular language describing the shared pattern of a set of strings. That is, a regular expression  $R$  represents a language  $L(R)$  over an alphabet  $\Sigma$ , where  $L(R)$  is a (possibly infinite) set of strings. The regular expression can be recursively defined by applying three types of operators to symbols  $\Sigma$ . The three types of operators are: *alternation*, *repetition*, and *concatenation* [177]. In the syntax of regular expression, the alternation operator is  $|$ , the common repetition operators are  $*$ ,  $+$ , and  $?$ . However, there is no special regex meta-characters required for the concatenation operator. Concatenating two regular expressions  $a$  and  $b$  is defined by putting  $b$  right after  $a$  [90].

The basic usage of regular expression is to check if a string conforms with the pattern described by the regex. If it does conform with the patter, the regex *matches* the string

and the string is called a *matching* string. Otherwise, the regex *mismatches* the string and the string is a *non-matching* string. The set of all the matching strings of a regex is also called the (semantic) *scope* of the regular expression. To decide whether a regular expression matches the string or not, the regex processor translate the regex into a deterministic finite automaton (DFA) or a non-deterministic finite automaton (NFA). A string is matched against the regex only if there is a way that the finite automaton can read the string and make state transitions from the *start state* to a *matching state* [41]. More details about regular expression matching are described in 4.2.1.

Regular expression is a fundamental approach of searching in many text-centric applications [34] and also plays a role in search engines. When searching the content of GitHub and StackExchange, queries can be constructed using regular expression [80, 193]. Regex filtering is also a common feature provided by Google services [147, 166]. But the regular expression used in the queries must be evaluated carefully to avoid the security vulnerabilities (i.e., ReDOS [43, 44]). For example, the regex is the culprit for the StackOverflow server outage in 2016 [141].

Regular expressions are used as constraints in data mining to find sequential patterns [62, 146, 190]. For example, the Sequence Mining Automata (SMA) is a type of Petri Net that constructed from the DFA of the given regular expression and its effectiveness in mining frequent sequences have been proven in experiments with multiple different datasets including a dataset of protein sequences [190]. Similarly, regular expressions are used to constrain pairwise sequence alignment algorithms [12].

Another area regular expressions are frequently seen and employed are security-oriented research. It is one of the techniques to detect SQL injection attacks [212] by improving the branch coverage of input-validation programs with the help of Reggae [109]. Regular expression matching is regarded as an essential functionality of modern intrusion detection systems (IDS) [202] for deep packet inspection. For example, DIDAFIT [105] is a database intrusion detection framework which summarizes the raw SQL queries into regular expressions and then uses these regexes to match against incoming queries.

The other areas where regular expressions are employed include lexical analysis [106], test case generation [7, 61, 63, 188], and string constraint solvers [94, 191].

## 2.2 Related Work

### 2.2.1 Regular Expression

#### 2.2.1.1 Regular Expression Security

One common misconception is that all regular expression languages are *regular languages* which can be represented using DFA, and so they are easy to model, easy to describe formally and execute in  $O(n)$  time. In fact, many regular expression matching engines run in exponential time in order to support useful features such as lazy quantifiers, capturing groups, look-aheads and back-references [131]. These features make regular expression engines sometimes inevitably to backtrack DFAs, which consumes more resources and takes more time. In the worst case, regular expressions can be exploited for denial-of-service (DoS) attacks, which is therefore called Regular Expression Denial of Service (ReDoS).

As a potential security vulnerability, some research has been focused on the security impact of ReDoS attacks. As a tool, ReScue [174] can craft regular expression attacks. One work statically analyzed regular expressions to check ReDoS attacks [97]. Studies have shown that ReDoS are common in programs [47], especially in javascript-based web servers [180].

#### 2.2.1.2 Regular Expression Comprehension

There are tens of thousands of bug reports related to regular expressions [179] and developers complain that they are complained to be hard to read and understand [31]. Further study [32] suggests that both the regex representation and the DFA size have impacts on regular expressions comprehension. To aid in regex creation and understanding, tools have been developed to support more robust creation [179], to allow visual debugging [22], or to help programmers complete regex strings [139]. Another study on the tools and strategies developers use during regular expression composition reveals that copied and pasted regular expressions need slightly modification to resolve compile errors and integrating visualization into the IDE helps developers compose the right regexes [124]. Other research has focused on removing the human from the creation process by learning regular expressions from text [13, 110].

### 2.2.1.3 Regular Expression Testing

Software test coverage can be measured at different levels of granularity, such as method, statement, branch, system, integration, module, and unit (e.g., [4, 108, 119, 152, 222]). Symbolic execution [6, 26, 27, 209] is one way to generate inputs and to obtain program test coverage at the level of branches. There are many tools for automated test generation [59, 142, 216]. For example, Reggae [109] aims to mitigate the large space exploration issues in generating test inputs for programs with regular expressions. Static analysis to reduce errors in building regular expressions by using a type system to identify errors like `PatternSyntaxException` and `IndexOutOfBoundsExceptions` at compile time [179].

### 2.2.1.4 Regular Expression Similarity

Regular expression similarity has been studied, but tool support is sparse. Rot, et al. proved regular expression language equivalence and the inclusion of DFAs that represent a subset of regular expression features [169], though no implementation is available. Dulucq, et al., defined tree edit distances [52] that can be applied to regular expression parse trees, which complements recent work proving that parse tree subsumption implies language subsumption [75]. Wang, et al. worked to cluster patterns by syntactic similarity [197] and others have worked to enumerate strings of regular expressions [122, 195]. These efforts toward computing and understanding regular expression similarity largely lack implementations.

Chapman and Stolee also studied regular expressions in Python [31]. They explored regular expression semantic similarity among projects [31], as well as the influential factors to the comprehension for regular expressions [33]. They also pointed out the top three language features appear in projects are ADD (one-or-more repetition with plus sign), CP (a capture group with parentheses) and KLE (zero-or-more repetition with star sign).

### 2.2.1.5 String Generations for Regular Expression

Rex [195] generates testing inputs for the regular expression according to its SFA representation. brics [129] generates inputs by traversing the DFA and building strings from the smallest bytes to the largest bytes of every DFA states. Some string generation tools need user-specified string length [63, 93, 129]. Some string solvers [93] and tools for generating testing inputs which use string solvers [63, 203] build finite-state automata based on string constraints.

EGRET [102] is focused on generating unexpected test strings to expose the regular expression errors, but it is based on common mistakes when creating regular expression rather than maximizing test coverage of regular expressions. MUTREX [9] employs distinguishing strings which can separate a mutated regular expression from the original one to expose system faults.

### **2.2.1.6 Regular Expression Generation**

Regular expression generation research includes algorithms and tools to generate test cases with regular expressions [8, 11, 189], generating regular expressions for DNA sequences [101] and matching patterns [30] [67] [137]. Enhancing regular expression pattern matching and processing speed is a common focus as well [15, 21, 24, 36]. The regular expression can be learned through genetic programming provided with a large number of labeled strings [18, 19, 110] Genetic programming has also been applied [36] to find equivalent alternative regular expressions which exhibit improved performances. For example, ReLIE [110] shows an algorithm of extracting regular expression from information and [13] presents another extraction algorithm for noisy unstructured text.

### **2.2.1.7 Regular Expression Performance and Optimization**

The study of regular expression optimization spans from the hardware, programming language, algorithm, to its usage in the source code. Study of regular expression matching hardware optimization is an active research area since the regex matching performance is essential in deep packet inspection (DPI) for network security. As pattern matching is performed on network devices, efforts have been made to make deterministic finite automata (DFA) more memory efficient [20, 56]. When nondeterministic finite automata (NFA) is preferred FPGA-based approaches are explored to boost performance by running multiple NFAs simultaneously [104, 127]. Hyperscan [202] is a highly optimized regex matching system for Intel architecture that can not only decompose regex matching into fast string matching and subregex matching but also support pattern matching parallelism with Single-Instructions-Multiple-Data (SIMD) -based algorithms. Hyperscan targets the usage of scanning big texts with thousands of regular expressions and more time is spent on compiling them. Thus it is not a good fit in scenarios where single regular expression is used or regular expressions being used change over time. Cody-Kenny and Fenton et al. [36] uses genetic programming to optimize regexes for runtime performance while maintaining their matching behaviors.

Unlike the above mentioned techniques for optimizing a fixed regular expression and varied input strings, other research has focused on expediting varied regular expressions processing on the fixed large bodies of text [17].

The performance of regular expression varies with different regex libraries [158, 159], regex engines [155], and programming languages [156, 157]. But very few work has been found on optimizing the regular expression usage from the granularity of source code. Chapter 3 lists a few frequent patterns used in optimizing regex usage in the code, but the effectiveness of those patterns is left unknown.

### **2.2.1.8 Regular Expression Tools**

Visualizations debugging tools [22, 140, 164] are developed to aid regular expression comprehension, and may provide some explanation for low test coverage of regular expressions in source code, that is, developers use online tools instead.

With respect to the finite automaton constructed from regular expressions, `brics` [129] contains a DFA implementation with very limited operations; while RE2 [41, 42] provides a DFA implementation which runs much faster than traditional regular expression engines in Java, Perl, and Python. Rex [195] builds a symbolic representation of finite automata (SFA).

Other regex tools include detectors of regex errors and regex optimizers. EGRET [102] generates both a set of accepted strings and a set of rejected strings for the given regex, and Spishak et al. [179] provides a type system to validate regular expression syntax at compile time. Regex-opt [162] and regexp-tree [Reg21] are regex optimizers that make regexes easier to read and reduce the risks of ReDoS. Complementary to these works, this body of work shows that incorrect regex semantics and compile errors are practical problems that developers need help with. It also shows 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.

## **2.2.2 Software Engineering Topics Related to This Dissertation**

### **2.2.2.1 Source Code Mining**

Mining properties of open source repositories is a well-studied topic. Exploring language feature usage by mining source code has been studied extensively for Smalltalk [28, 29],

JavaScript [168], Python [31], and Java [53, 71, 111, 144], and more specifically, Java generics [144], Java reflection [111], and Python regular expressions [31]. One study of source code evolution focuses on several popular open source programs and identifies the code changes through AST matching [135]. However, the prior work in regular expression mining looked at the most recent version of the regular expressions, not at their edit histories.

The *code churn* metric represents the number of lines of code that were added/deleted or changed. This concept is was introduced as a means to measure the impact of code changes [133] and has been used to predict software failures and defects [64, 79, 98, 123, 134, 176]. The classic definition of code churn is the absolute value of a code delta between two sequential builds. For regular expressions, we borrow the concept of code churn and measure regular expression evolution by calculating the Levenshtein distance. We do not, however, have a measure of regular expression faultiness to tie together distance with fault-proneness; exploring that is left for future work.

#### **2.2.2.2 Language Features**

The language features and their usage were explored in diverse prior work by mining source code in programming languages, such as Java [53] [144], JavaScript [168]. It is common for the programming language to have new language features as it evolves, and Dyer et al. investigated how these new features are adopted by programmers once released and the most frequently used features [53]. They concluded that the adoption rate of Annotation Use is exceptionally high and about half of the usage is `@Override` annotation. Generic Variable declarations are popular among developers as well, and the top three types are `List`, `ArrayList` and `Map`. These results are consistent with the work of Parnin et al [144].

#### **2.2.2.3 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 [49, 115, 184, 217, 220]. GitHub pull requests [69, 70, 118, 170] 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 [78, 116, 138]. Some research targets emerging applications, such as TensorFlow bugs [217] and Blockchain bugs [196], while others target distributed systems such as node change bugs [113] and concurrency bugs [114]. More specific bug types include bugs in exception-related code [37], bugs in patches [213], bugs in test code [194], numerical bugs [49], string-related bugs [54], performance bugs [172], and cross-project correlated bugs [115]. Our study joins this list with its focus on regex-related bugs. It is interesting to see that 37% of the string-related bugs [54] 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 [49, 114, 172, 213, 217], bug patterns [112, 114], and fix strategies [114, 115, 172]. Tan, et al. [184] conduct a temporal analysis to study the trend of different types of bugs with software evolution. Zhong and Su [220] evaluate the differences between bug fixes by programmers and the fixes by automatic program repair. Selakovic and Pradel [172] measure the complexity of optimization code changes. Wan et al. [196] 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.

#### **2.2.2.4 Code Smells and Design Smells**

Smells were introduced by Fowler [58]. Code smells and design smells imply that certain structures in the code or design need refactoring. Researchers have studied the impact of code smells on program comprehension [1, 51], finding that the more smells in the code, the harder the comprehension. Code smells have been extended to other language paradigms including end-user programming languages [76, 77, 182, 183]. Using community standards to define smells has been used in refactoring for end-user programmers [182, 183]. There is also lots of research focused on code refactoring to categorize or detect code smells and design smells in source code [128, 143, 173] and to understand the mutual impact between those bad smells and the software development process [92, 192].

#### **2.2.2.5 API and Library Usage.**

Our study in Chapter also have overlaps with research on library and API usages. The library usage studies have explored the motivation of library migration [87], how to choose

the best library candidate [126], which version should be used [126], and how to automate the migration of the underlying API library from one to another [89, 185, 207]. The API usage studies have explored mining API usage patterns [221], detecting API misuses [2, 3, 16, 73, 91], studying API evolution [50, 95, 175, 218], and automatic refactoring for API changes [148, 187] via documentation [50, 99, 175, 218], source code and its change history [50, 187], and bytecode [148]. The majority of API-misuse detectors are via static analysis and suffer from low precision and recall in practice [3, 73] while Catcher [91] 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 [73]. 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 [126] or to identify and predict API usage trends over time [125].

#### **2.2.2.6 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. [215] and Marsavina, et al. [121] study the testing strategies and the co-evolution patterns between production code and test code evolution. Pinto, et al. [151] 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. [69] 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. [100] study the distribution of test cases across 50,000 GitHub projects. Pham, et al. [149] 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 [70] 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 [74, 86, 117, 171, 206].

## CHAPTER

# 3

## REGULAR EXPRESSION BUGS

Chapter 3 presents a study regular expression bugs and uncover the nature of the issues that relate to regular expressions, in particular, the nature of the issues that developers end up addressing. It further explores the role of test code in bug fixing for regex-related bugs. Portions of this work were published in *MSR 2020* [200].

### 3.1 Introduction

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 [45], 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 [199], 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 most important findings in this chapter are as follows:

- Incorrect regular expression semantics are 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%).
- 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.
- 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.
- 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.

## 3.2 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 3.4).

**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 [70]: 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 3.5).

**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 3.6).

## 3.3 Study

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

### 3.3.1 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 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.

#### 3.3.1.1 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 [186], Mozilla [130], Google [68], and Facebook [55]. Using the GitHub GraphQL API [65], we searched for merged pull requests<sup>1</sup> 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 [66]. This resulted in 664 merged pull requests from 195 GitHub repositories in the 4 organizations.

#### 3.3.1.2 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,

---

<sup>1</sup>While we avoid many perils of mining GitHub [88] 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 3.8.

```

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)))));

```

**Figure 3.1** Example of Regex Addition from a pull request in JavaScript (mozilla/zam-boni#442).

**Table 3.1** 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)

Figure 3.1 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).

### 3.3.1.3 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 3.1 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 differences before and after the PR, hence avoiding issues from reworked commits (Peril VII [88]). 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 [5].

### 3.3.2 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.

### 3.3.2.1 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 3.5), regular expressions can look different but have the same semantics [32] (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 3.5 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.

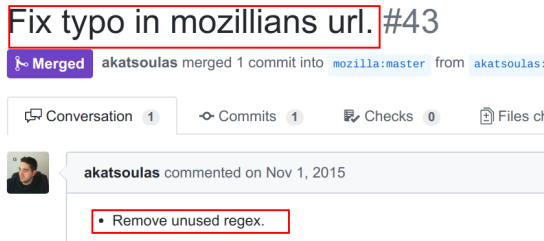
### 3.3.2.2 Smells and Bug

Code smells refer to code that has correct semantics but is flawed in another way, such as poor maintainability, performance, or readability [32]. 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* [58]. The well-known design smells include *spaghetti code* and *stovepipe system* [25]. 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 work, 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.

### 3.3.3 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 3.2 shows three illustrative example PRs. In Figure 3.2a, PR mozilla/feedthefox#43 addresses two problems. One is a typo of

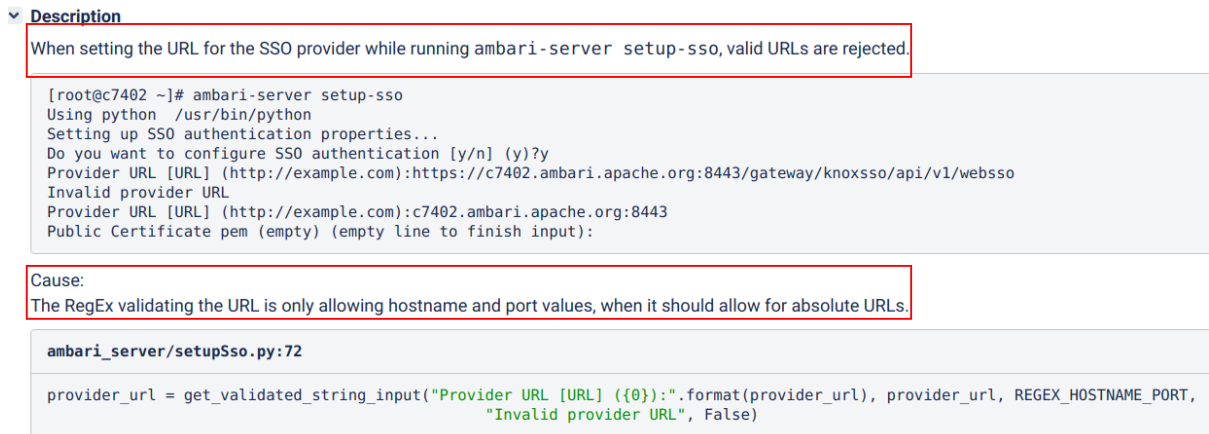


(a) PR mozilla/feedthefox#43.

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



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



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

**Figure 3.2** Examples to illustrate identifying problems addressed in pull requests.

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 3.2b 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*.<sup>2</sup> Figure 3.2c 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

<sup>2</sup>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 3.2.

### 3.3.4 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.

#### 3.3.4.1 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 [70] (*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 [153] library, which provides APIs to retrieve GitHub resources. The *allPRs* dataset [70] 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 3.3), we used the non-parametric Mann-Whitney-Wilcoxon Test [120] to investigate whether our dataset, *regexPRs*, and the *allPRs* dataset have the same distribution. These comparison results are presented in Table 3.3.

#### 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 ( $R_{rm}$ ),

```

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()) {

```

**Figure 3.3** Example of Regex API Changes from a pull request in Java (google/Exo-Player#3185).

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 Figure 3.1 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 3.1, 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 [198].

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 Figure 3.1 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 Figure 3.3 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).

### 3.3.4.3 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 3.2). 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 3.5).

## 3.3.5 RQ3 Analysis: Test Code Characteristics

To answer RQ3, we identified the files used for test code<sup>3</sup> and then manually counted the test cases being changed in the PR. While rare, if a PR contained more than one regex-related bug (six PRs in our dataset), we mapped the test code to the appropriate regular expression.

### 3.3.5.1 Identifying Test Code Files

Test code files are usually located under “`test`” (or “`tests`”, “`__test__`”) and “`spec`” (or “`specs`”) directories [70, 132]. Following certain naming conventions, the test file names often contain the word “`test`” or “`spec`” as prefix or suffix [84, 85, 154]. Therefore, we regarded any files included in the PR which satisfy either of the two conditions as a candidate test code file.

---

<sup>3</sup>*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.

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.

### 3.3.5.2 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 [23, 100, 151] are also used more often than test files and test classes [215].

The process for test case identification depended on the language and testing framework used. In the Python unit testing framework `unittest`<sup>4</sup> or `pytest`,<sup>5</sup> test method names start with the letters “`test`”. In the Java Junit framework,<sup>6</sup> 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`,<sup>7</sup> `jasmine`,<sup>8</sup>) 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.

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 3.3.4.2).

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

---

<sup>4</sup><https://docs.python.org/3/library/unittest.html>

<sup>5</sup><https://docs.pytest.org/en/stable/>

<sup>6</sup><https://junit.org/junit5/>

<sup>7</sup><https://mochajs.org/>

<sup>8</sup><https://jasmine.github.io/>

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 [151]. 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*.

### 3.4 RQ1 Results: Bug Categories

As is done in prior work on categorizing software bugs, we identify the *root cause* and *manifestation* of the bugs [49, 114, 172, 184, 217]. 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 3.2. When the root cause is the *regex*, the regex itself causes an issue; examples include incorrect semantics<sup>9</sup>, 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

---

<sup>9</sup>In this paper, we correct the naming of one of the manifestations from the original paper [200]. In the original paper, we use the term, *incorrect behavior* but it should be *incorrect semantics*, per the definitions in Section 3.3.2.

**Table 3.2** 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%)			
	Bad Smells	Design Smells	Alternative regex API		2 (7.4%)	33 (9.3%)	
			Unnecessary computation		9 (33.3%)		
		Code Smells	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%)
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%)	

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).

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 3.2). 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 3.2). 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.

### 3.4.1 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*.

#### 3.4.1.1 Regex: Incorrect Semantics

*Incorrect Semantics* is the dominant manifestation for bugs with the regex as the root cause (75.7%, 165/218). Table 3.2 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 [124] and tend to expand the scope of regular expressions as software evolves [198].

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.

#### 3.4.1.2 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*.

#### 3.4.1.3 Regex: Bad Smells

The regex *bad smells* we observed can be divided into two categories, as shown in Table 3.2: *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 [47] vulnerability<sup>10</sup>). *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".<sup>11</sup> 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.

### 3.4.2 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,

---

<sup>10</sup>Since ReDoS cares about the time complexity of running the regular expression, we regard it as a performance issue.

<sup>11</sup><https://eslint.org/docs/2.0.0/rules/no-regex-spaces>

9.3%) refer to the incorrect regex API usage manifesting as either *incorrect computation* (6, 1.7%) or *bad smells* (27, 7.6%).

### 3.4.2.1 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`). 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.

### 3.4.2.2 Regex API: Bad Smells

We found 27 PR bugs that stem from *bad smells* in using regex APIs. Table 3.2 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,<sup>12</sup> when the regex to compile is not hard-coded and unknown to the API method

---

<sup>12</sup>In Section 3.4.1.2 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.

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.<sup>13</sup> Similarly, when the `flags` argument is deprecated<sup>14</sup> in JavaScript regex APIs, `input.replace('<', '&lt;')`, `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 improper exception handling (8). Additional PRs reduce the number of calls to the regex API in the interest of performance (9).

### 3.4.3 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 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

---

<sup>13</sup><https://jakarta.apache.org/oro/>

<sup>14</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String/replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace)

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).

#### 3.4.3.1 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 un-sanitized log messages, and sanitizing them is a new feature. Table 3.2 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).

#### 3.4.3.2 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).

#### 3.4.3.3 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.

**Table 3.3** Comparing selected features of regex-related PRs (*regexPRs*) to merged PRs (*allPRs*) from prior work [70].

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

## 3.5 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.

### 3.5.1 Complexity of Regex-related PR Fixes

Given the prevalence and severity [47, 179] 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 [70]. Table 3.3 shows the pull request feature distributions for our dataset (*regexPRs*) and the merged PRs from prior work (*allPRs*), as described in Section 3.3.4.1. 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-Wilcoxon test [120] 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 [145, 178].<sup>15</sup> 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

<sup>15</sup>The skewness score for normal distributions is zero.

**Table 3.4** 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 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)	

*regexPRs*, because of which the 95% percentile of *num\_commits* in *regexPRs* is smaller than that in *allPRs*.

As shown in Table 3.3, *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 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.

### 3.5.2 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 3.4 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.

**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 3.2), 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 3.5). 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 3.5). 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.

### 3.5.3 Recurring Patterns to Fix Regular Expression Bugs

Table 3.5 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`).

**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

**Table 3.5** 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: <code>regex="a.png"</code> After: <code>regex="a\\.png"</code>
2	Extend or shrink the character class	17	Before: <code>value_regex = r'[_\w]+'</code> After: <code>value_regex = r'[_\-\w]+'</code>
3	Replace regex with string methods	15	Before: <code>if re.match(".*error.*",message):</code> After: <code>if "error" in message:</code>
4	Replace regex with existing parser	11	Before: <code>EMAIL_REGEX_PATTERN.matcher(email).matches();</code> After: <code>import javax.mail.internet.InternetAddress;</code> <code>InternetAddress emailAddr = new</code> <code>↳ InternetAddress(email);</code> <code>emailAddr.validate();</code>
5	Add or remove a regex alternation	10	Before: <code>regex="win32 windows"</code> After: <code>regex="wind32 windows win64"</code>
6	Add or remove a regex to the regex list	9	Before: <code>'regexes': [</code> <code>re.compile('Ubuntu HW 12.04 x64 .+')</code> <code>]</code> After: <code>'regexes': [</code> <code>re.compile('Ubuntu (ASAN )?HW 12.04 x64 .+'),</code> <code>re.compile('^Android 4\.2 x86 Emulator .+'),</code> <code>]</code>
7	Correct type of regex representation Language: Python	6	Before: <code>'pattern': '\d{1,2}/\d{1,2}'</code> After: <code>'pattern': r'\d{1,2}/\d{1,2}'</code>
8	Checking null values for regex	5	Before: <code>Matcher matcher = regex.matcher(currentLine);</code> After: <code>Matcher matcher = currentLine == null ? null</code> <code>↳ : regex.matcher(currentLine);</code>
9	Regex static compilation	4	Before: <code>String BLACKLIST = "...";</code> <code>boolean method(String name) {</code> <code>return !(name.matches(BLACKLIST));</code> <code>}</code> After: <code>Pattern BLACKLIST = Pattern.compile("...");</code> <code>boolean methodE(String name) {</code> <code>return</code> <code>↳ !(BLACKLIST.matcher(name).matches());</code> <code>}</code>
10	Conditional checking before regex execution	4	Before: <code>Matcher</code> <code>↳ m=Pattern.compile(regex).matcher(currentLine);</code> After: <code>if currentLine.contains("error"){</code> <code>Matcher m =</code> <code>↳ Pattern.compile(regex).matcher(currentLine);</code> <code>}</code>

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 `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 3.5 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.

## 3.6 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 3.6** 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 %)		

### 3.6.1 Overview

Table 3.6 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<sup>16</sup> 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 [69]. 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 [149].

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 3.6). 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 [151] 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 [151] are 56%, 29%, and 15%. Those percentages in regex-related PRs are 64%, 30%, 7%. The study on test case changes [151] 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

<sup>16</sup>We lost the data for three bugs between the original paper [200] and this analysis of the test code. We know there were test code changes, but the details are no longer available.

**Table 3.7** 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 API	Incorrect Computation	2	2	0	4/6 (66.67%)
	Bad Smells	6	1	1	7/27 (25.93%)
Other Code	New Feature	40	9	2	44/59 (74.58%)
	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%)

having test code change.

### 3.6.2 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 3.7, PRs for solving regex *compile errors* do not involve any 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

**Table 3.8** 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

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

### 3.6.3 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 3.8 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 3.7, *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 [151] 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 [199]. Therefore, when an invalid string is accepted, developers have the option to delete the invalid string or to edit the string into a 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.

## 3.7 Discussion

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 [45] and porting regular expressions causes semantic and performance differences [46]. 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 3.4.2.2). 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 3.5).

**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 [160, 165, 204] 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 [124]. Further research efforts are needed to better understand when to use and when not to use regexes (Section 6.2.4).

**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 3.4.2), improper exception handling (Section 3.4.2.2), and unreadable or inefficient regexes (Section 3.4.2.2). 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 [31], cannot determine if a regex is compiled too often (Pattern 9, Table 3.5), if a string library would be more appropriate (Pattern 3, Table 3.5), or if a meta-character should be escaped (Pattern 1, Table 3.5). While helpful for understanding matching behavior, developers could benefit from tool support within the IDE that can consider the context (Section 6.2.4).

**Regex performance is about more than regex complexity.** Prior work on regexes and ReDoS [47, 174, 181, 208] 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 3.5) 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 [162, Reg21] 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 [161]. 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 [199] 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 [74, 117].<sup>17</sup> 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 [199] 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

---

<sup>17</sup>The former reports 94% of found bug-fixing patches contain test cases and the latter reports only four patches with no tests changed.

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.<sup>18</sup> PR [apache/cxf#479](#) also contains a benchmark to compare the execution 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 [199] 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.

### 3.8 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 [214]. For the labelled ones, previous study haven shown that misclassification occurs frequently between bugs and non-bugs [150]. 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 [88]. 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.

---

<sup>18</sup><https://openjdk.java.net/projects/code-tools/jmh/>

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 [38–40], 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 [215], 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 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 [220]. 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 [70], 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.

### 3.9 Summary

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 chapter presents a comprehensive study of 356 merged regular expression related pull requests from 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.

## CHAPTER

# 4

# REGULAR EXPRESSION TESTING

Chapter 4 explores how thoroughly tested regular expressions are by examining open source projects. Portions of this chapter were published in *FSE 2018* [199].

## 4.1 Introduction

Traditional code coverage criteria, are rather coarse-grained when it comes to regular expressions. Statement coverage requires the regular expression to be invoked at least once. If the regular expression call site appears in a predicate, branch coverage requires that the regular expression is tested with at minimum two strings, one in the language of the regular expression and one not. However, these metrics ignore the complex structure represented by a regular expression. We propose to use test metrics for graph-based coverage [4] over the DFA representation of regular expressions.

Regular expression tools can help support developers in their creation and testing of regular expressions. These tools either automatically generate strings according to the given regular expressions [93, 94, 129, 195] or automatically generate regular expressions according to the given list of strings [13, 110]. Rex [195] is a tool for analyzing regular

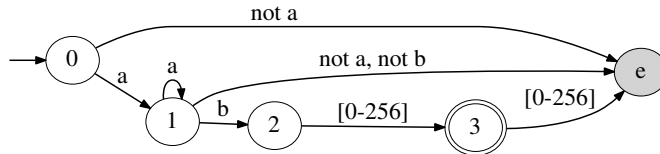
expressions through symbolic analysis. Given a regular expression  $R$ , Rex uses the Z3 [48] SMT solver to generate members of the language by treating it as a satisfiability problem. Like automatic test case generation tools, integrating these generated results into software testing can help automate the process, but it is not clear how well covered the regular expressions would be compared to developer-written tests.

In this chapter, we focus on empirically measuring how well tested regular expressions are and further explore the potential for using existing tools, specifically Rex, to improve the test coverage. First, we measure the test coverage of regular expressions in the wild based on a set of 1,225 Java projects on GitHub containing 15,096 tested regular expressions. Second, we measure the test coverage of strings generated by Rex and compare the coverage achieved against the strings generated by developers in the GitHub projects. Our main findings are:

- Of 18,426 call sites for three pattern matching API methods identified statically in 1,225 GitHub projects, only 3,093 (16.8%) are ever executed by test suites (RQ1).
- Of 15,096 regular expressions captured during test suite execution of 1,225 GitHub projects, 10,970 (72.7%) use only failing inputs (4,941) or only matching inputs (6,029) (RQ1).
- The Rex-generated test inputs achieve similar coverage levels to the developer-written tests (RQ2).

## 4.2 Motivation

In this chapter, we explore test coverage metrics over the DFA representing a regular expression. This requires three informal explorations to ensure feasibility and assess the potential impact. First, we introduce the concepts and terms related to DFA in regular expression matching. Then, we explore the potential of building DFAs from regular expressions by analyzing regular expressions collected from an existing Python dataset [31] and testing them for regularity [177]. Next, we show intuitively how existing coverage metrics are insufficient. Finally, to motivate the structural coverage metrics, we explore whether faults can lie along untested paths in a DFA.



**Figure 4.1** Forward FullMatch DFA of regular expression ‘a+b’.

## 4.2.1 DFA with Regular Expression Matching

### 4.2.1.1 DFA and NFA

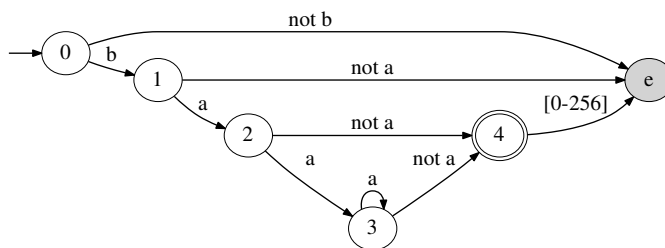
For a given language, there are many regular expressions that can describe it. A regular expression can be represented as a string of tokens, a finite state automaton in deterministic (DFA) form, or in non-deterministic (NFA) form. The regex processor, in order to match the regex against a string, translates the regular expression into deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA). For a given input, a state in DFA can only transition to only one other state while a state in NFA has multiple states to transition to and requires backtracking so that it can try other states after previous states tried lead to non-matching state.

Although theoretically the DFA and the NFA are equivalent representation of a same regex, some regular expressions cannot be converted into a DFA. This is due to that regular expression libraries in current programming languages are more expressive than regular languages. For example, the backreference feature cannot be modeled by a finite-state automaton and requires a push-down automaton. When creating support techniques and tools for regular expression testing, regularity, or how regular are regular expressions, is important since it defines the abstractions we can use for test coverage.

In this research DFA is chosen for measuring testing coverage of regular expression not only because DFA are comparatively simpler than NFA but also because the DFA provide a stable representation at static but NFA graph structure can only be depicted during matching process.

### 4.2.1.2 RE2

RE2 [41, 42] is a regular expression processing engine developed by Google with DFA implementations. Figure 4.1 shows one DFA of the regular expression a+b built using RE2, and we take this opportunity to describe the DFA notation used throughout this work.



**Figure 4.2** Backward FullMatch DFA of regular expression ‘a+b’.

Node 0 is the *start state*, indicated by the incoming arrow. Nodes with double circles are *matching states*, such as Node 3. Node e is the *error state*, denoting a mismatch. The edges are labeled with transitions, often using syntactic sugar for ease of interpretation. The edge  $\overrightarrow{01}$  is traversed when a character ‘a’ is read. If any other character is read at Node 0, (i.e., **not a**), edge  $\overrightarrow{0e}$  is traversed. There is a self-loop on Node 1 for the character a. If character ‘b’ is read from Node 1, then edge  $\overrightarrow{12}$  is traversed.

In RE2, when reading an input string, byte [256], is added as a text-end marker. For example, the input string “aab” is transformed to the byte stream [97 97 98 256], as [97] is the byte for ‘a’, [98] is for ‘b’, and [256] marks the end of the string. Byte [256] is matched on edges ‘[0-256]’, ‘not a’, or ‘not a, not b’.

### 4.2.1.3 DFA Types

There are different types of matching between a regular expression and a string. The Java function `Pattern.matches` requires the regular expression to match a string from its beginning to its end; Python’s `re.match` requires the regular expression to match a string only from its beginning, not necessarily match to the end of the string; and the C# function `Regex.Match` requires the regular expression to match only a substring of the input string. These are called *FullMatch*, *FirstMatch*, and *ManyMatch*, respectively.

Given a regular expression and an input string to match, we could build multiple DFAs with different considerations. We can build a Forward DFA and Backward DFA depending to the direction of scanning the regular expression. Figure 4.1 and Figure 4.2 shows the differences between a Forward FullMatch DFA and a Backward FullMatch DFA. For the regular expression "a+b", its Forward DFA (Figure 4.1) and Backward DFA (Figure 4.2) are different in both DFA size and DFA structure if the regex is not a syntactic palindrome. We could build a static DFA with a regular expression alone or build a DFA on-the-fly (dynamic DFA) during a matching between a regular expression and an input string. In this work, we choose the Forward static DFA for the regular

expression and the Forward dynamic DFA for *FullMatch* of the regular expression and the input string.

### 4.2.2 How Regular Are Regular Expressions?

Regular expressions in source code can contain non-regular features, such as backreferences. An example is the regular expression `([a-z]+\1)`, which matches a repeated word in a string, such as "appleapple". Building a DFA is not possible for this since this regular expression is non-regular. For regular expressions in source code that are indeed regular, we can build DFAs and measure coverage based on a test suite. Here, we are testing how many of the regular expressions in the wild are truly regular.

We explore an existing and publicly available dataset of 13,597 regular expressions scraped from Python projects on GitHub. To test for regularity, we use an empirical approach since the ability to build a DFA from a regular expression implies that it is regular [177]. Of the 13,597 Python regular expressions, 13,029 (95.9%) are regular in that we were successful in building DFAs for each using the RE2 [42] regular expression processing engine. For the remaining 568, we investigated each by hand. One regular expression was removed because its repetition exceeds the RE2 limits. While it may indeed be regular, to be conservative, we mark it as non-regular. An additional 81 contained comments within the regular expressions, which are unsupported in RE2, so these were also assumed to be non-regular; 128 contained unsupported characters. The remaining 368 were non-regular as they contained backreferences.

In the end, with nearly 96% of the regular expressions being regular (as a low estimate), we conclude that most regular expressions found in the wild are regular and thus can be modeled with DFAs.

### 4.2.3 Limitations of Code Coverage

Statement coverage requires the regular expression to be invoked at least once. If the regular expression call site appears in a predicate, branch coverage requires that the regular expression is tested with at minimum two strings, one in the language of the regular expression and one not. However, these metrics .

In this research, we posit that code coverage metrics [4] [108, 152, 222] such as statement, branch, and path, are too coarse-grained for regular expressions. Statement coverage requires that the code containing the regular expression is reached, leading to a

```

1  if (Pattern.matches("-d|--data", strInput)) {
2      System.out.println("YES");
3      ...
4  } else {
5      System.out.println("NO");
6      ...
7  }

```

**Figure 4.3** Example to explicit the limitation of code coverage.

minimum of one test input for the regular expression. If the regular expression is in a statement where the control flow is dependent on the matching outcome, branch coverage requires that the regular expression have at least two inputs, one that evaluates to true and another that evaluates to false.

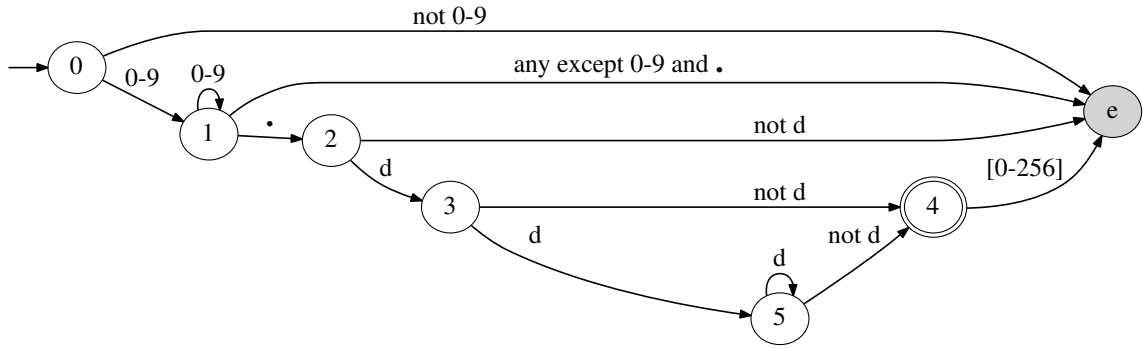
Consider the Java code snippet in Figure 4.3. The *call site* for method `Pattern.matches` is on line 1. The regular expression is `-d|--data`. Statement coverage of the regular expression requires that line 1 is executed and branch coverage requires two test inputs, one to cover the `true` branch and one to cover the `false` branch. Using coverage metrics based on the DFA representation of the regular expression, on the other hand, would require 1) each branch to be covered, and 2) each *case* in the regular expression, “-d” and “--data”, to be covered. Such metrics measure test coverage of the regular expression’s control flow (i.e., the DFA) just like branch coverage measures test coverage of source code’s control flow graph.

Existing tools and techniques can direct test input generation toward areas of untested paths. One technique among these is symbolic execution [6, 27, 63, 96, 109], and Rex [195] has been developed for symbolic analysis of regular expressions. However, Rex focuses solely on the matching behavior [195], which limits its ability to cover the false branch in the Java example above. Brics [129] and Hampi [93, 94] similarly only generates passing strings. While useful, there are no guarantees of structural coverage.

#### 4.2.4 DFA Coverage Example

Bug reports related to regular expressions abound. A search for “regex OR regular expression” in GitHub yields over 555,000 issues, with 22% of those still being open. One in particular illustrates how coverage metrics on the DFA could have brought a particular bug to the developer’s attention sooner. This bug report<sup>1</sup> describes an issue with the

<sup>1</sup><https://github.com/maven-nar/nar-maven-plugin/issues/228>



**Figure 4.4** Full-match DFA for regular expression: `\d+\.\d+`.

regular expression `\d+\.\d+` in the NAR plugin for Maven. Figure 4.4 shows the DFA of this regular expression built using RE2 [42], and we take this opportunity to describe the DFA notation used throughout this chapter.<sup>2</sup>

Node 0 is the start-state, indicated by the incoming arrow. Nodes with double-circles are accept states, such as Node 4. Node e is the error state, denoting a mismatch. The edges are labeled with transitions, often using syntactic sugar for ease of interpretation. The edge  $\overrightarrow{01}$  is traversed when a digit from 0-9 is read. If any other character is read at Node 0, (i.e., `not 0-9`), edge  $\overrightarrow{0e}$  is traversed. There is a self-loop on Node 1 for digits 0-9. If the period character is read from Node 1, then edge  $\overrightarrow{12}$  is traversed.

In RE2, when reading an input string, byte [256], is added as a text-end marker. For example, the input string “0.0” is transformed to the byte stream [48 46 48 256], as [48] is the byte for ‘0’, [46] is for ‘.’, and [256] marks the end of the string. Byte [256] is matched on edges ‘[0-256]’, ‘not 0-9’, ‘not d’, or ‘any except 0-9 and .’.

The bug report mentions that the regular expression `\d+\.\d+` is buggy and the patch adds an escape before the second d, `\d+\.\d+`. The intended behavior is to match input strings with one or more digits, followed by a period, followed by one or more digits.

In this research, the structural metrics could reveal this fault. With the DFA in Figure 4.4, when Node 3 is reached, the fault may be revealed. Input “0.d” traverses  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  and ends in an accept state, when it should fail. However, input “0.d3” traverses  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow e$  and ends in an error state, as expected. Covering edge  $\overrightarrow{2e}$  may also reveal the fault; input “2.3” traverses  $0 \rightarrow 1 \rightarrow 2 \rightarrow e$  and ends in an error state, when it should be accepted. Requiring coverage of all feasible nodes and edges could have revealed this fault in the regular expression.

<sup>2</sup>The regular expression in the bug is triggered by `Matcher.find()` with a `ManyMatch` DFA. For simplicity, we show the `FullMatch` DFA, a subgraph of the `ManyMatch`.

As with code coverage, uncovered artifacts alert the programmer to untested behavior. Such coverage information can indicate that a regular expression is not well tested and for some inputs it may not behave as intended, as is the case here.

## 4.3 Regular Expression Test Coverage Metrics

We explore fine-grained coverage metrics for regular expressions based on a DFA representation. The intuition is that since regular expressions are equivalent to DFAs [177], and 96% of regular expressions in the wild were found to be regular (Section 4.2.2), then graph coverage metrics over the DFA can be used to test the behavior within the regular expression. We discuss three levels of coverage: Node Coverage (NC), Edge Coverage (EC), and Edge-Pair Coverage (EPC). These coverage metrics are adopted from graph coverage metrics proposed by Ammann and Offutt [4].

### 4.3.1 Graph Notation

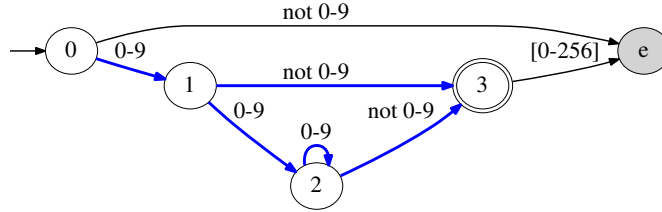
For ease of exposition, we expand on the traditional definition of a DFA. In this research, a DFA graph  $G = \{N, N_0, N_m, N_e, E\}$  where:  $N$  is the set of all nodes,  $N_0$  is the initial node,  $N_m$  is the final matching/accept node,  $N_e$  is the final failing/error node, and  $E$  is a set of all edges. For the DFAs in this research, there is only one initial state, one accept state, and one error state.<sup>3</sup>

The states in a DFA are the nodes  $N = \{n_0, n_1, \dots, n_k\}$ . For any two nodes  $n_1$  and  $n_2$  such that  $\{n_1, n_2\} \subseteq N$ , if there is a transition from  $n_1$  to  $n_2$  in DFA, then the edge  $\overrightarrow{n_1 n_2} \in E$ ; the start and end-state of the path may be the same node, as is the case of self-loops. Edge pairs are defined by paths of length two in the DFA. For example, if  $\{\overrightarrow{n_1 n_2}, \overrightarrow{n_2 n_3}\} \in E$ , we denote the edge pair as  $\overrightarrow{n_1 n_2 n_3}$ . In the case of self-loops,  $\overrightarrow{n_2 n_2 n_2}$  is also a valid edge-pair.

Given an input string and a regular expression, the initial node  $N_0$  is visited first. Transitions are taken as each character in the string is consumed. The result of the matching process ends in either the accept node  $N_m$  or the error node  $N_e$ . In standard DFAs, a traversal can end in any node. However, the DFA generation algorithm used in this research is based on the RE2 tool, which always ends processing in an explicit matching/accept ( $N_m$ ) or error ( $N_e$ ) state. In this tool, given a regular expression and an

---

<sup>3</sup>In a FullMatch DFA, there could be several matching nodes, and only one accept. In this work it is simplified to use only one accept state.



**Figure 4.5** Full-match DFA from RE2 [42] for the regular expression  $\backslash d^+$ . RE2 interprets every string as a byte stream; the range of bytes is  $[0-256]$  where  $[256]$  is added to mark the end of a string. Thus, the input string “2” would be represented as  $[50\ 256]$  and traverse the following path:  $0 \rightarrow 1 \rightarrow 3$ . The edges marked  $0-9$  represent the byte range  $[48-57]$ ; edges  $not\ 0-9$  represent the byte ranges  $[0-47]\ [58-256]$ .

**Table 4.1** Coverage of  $\backslash d^+$ :  $S = \{“2”, “1001”, “u”, “100u0”\}$ ,  $S_{succ} = \{“2”, “1001”\}$ , and  $S_{fail} = \{“u”, “100u”\}$ .

	$S$	$S_{succ}$	$S_{fail}$
$NC$	100.0%	80.0%	100.0%
$EC$	100.0%	71.4%	85.7%
$EPC$	75.0%	62.5%	50.0%

input string, the input string is interpreted as a byte stream, with byte  $[256]$  added to the end to mark the end of the string. Thus, an input string “2” would be interpreted as  $[50\ 256]$  and the input string “1001” would be interpreted as  $[49\ 48\ 48\ 49\ 256]$ .

As a running example, consider regular expression  $R = \backslash d^+$  and graph  $G$  in Figure 4.5. In  $G$ ,  $N = \{0, 1, 2, 3, E\}$ ,  $N_0 = 0$ ,  $N_m = 3$ ,  $N_e = e$ ,  $E = \{\vec{0e}, \vec{01}, \vec{12}, \vec{13}, \vec{22}, \vec{23}, \vec{3e}\}$ , and  $EP = \{\vec{012}, \vec{013}, \vec{122}, \vec{123}, \vec{13e}, \vec{222}, \vec{223}, \vec{23e}\}$ . Edges  $0-9$  cover bytes  $[48-57]$ , and edges  $not\ 0-9$  cover the byte ranges  $[0-47]\ [58-256]$ ; we use the decimal representation to improve clarity.

At this point, note that this is not the smallest DFA for the regular expression  $\backslash d^+$ . As the same tool is used for the construction of all the DFAs, any impact of the DFAs not being minimal (e.g., extra nodes or edges compared to the minimal representation) is distributed throughout the whole data set and consistent across all experiments. While we refer to RE2 [42] for full details of the DFA construction, though some intuition is provided in Section 4.5.2.2.

### 4.3.2 Coverage Criteria

Given a set of strings  $S$  and a DFA  $G$ , for all  $n \in N$ , we mark  $n$  as *covered* if  $n$  is visited during the processing of some  $s \in S$ . Similarly, edges  $e \in E$  and edge-pairs  $ep \in EP$  are marked as *covered* if they are traversed during the processing of some  $s \in S$ . The sets

of covered nodes, edges, and edge-pairs are denoted  $N_{cov}$ ,  $E_{cov}$ , and  $EP_{cov}$ , respectively. These sets are aggregated over all  $s \in S$ .

As defined in prior work [4], we adopt coverage definitions for node coverage ( $NC$ ), edge coverage ( $EC$ ), and edge-pair coverage ( $EPC$ ) as follows:

**Definition 1 (Node Coverage %)**  $NC = 100 \times \frac{|N_{cov}|}{|N|}$

**Definition 2 (Edge Coverage %)**  $EC = 100 \times \frac{|E_{cov}|}{|E|}$

**Definition 3 (Edge-Pair Coverage %)**  $EPC = 100 \times \frac{|EP_{cov}|}{|EP|}$

To illustrate the coverage levels, consider the graph  $G$  for the regular expression  $\backslash d+$  in Figure 4.5 and the string  $s_0 = "2"$  with  $S = \{s_0\}$ . Traversing  $G$  visits  $0 \rightarrow 1 \rightarrow 3$  (recall that "2" is interpreted as the byte stream [50 256]). Node 3 is the accept node, which denotes that the regular expression matches the input string (i.e.,  $s \in L(R)$ ). During the traversal of  $G$ , nodes  $\{0, 1, 3\}$  are visited, meaning that  $N_{cov} = \{0, 1, 3\}$ ,  $E_{cov} = \{\overrightarrow{01}, \overrightarrow{13}\}$ , and  $EP_{cov} = \{\overrightarrow{013}\}$ . The coverage levels for  $\backslash d+$  by input strings  $S = \{s_0\}$  are:  $NC = 60\% (3/5)$ ,  $EC = 28.6\% (2/7)$ , and  $EPC = 12.5\% (1/8)$ .

Next, consider adding the string  $s_1 = "1001"$ , which is interpreted as the byte stream [49 48 48 49 256]. Now,  $S = \{s_0, s_1\}$ . Traversing  $G$  on  $s_1$  traverses the following path:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 3$ , adding node 2 to  $N_{cov}$ , edges  $\overrightarrow{12}$ ,  $\overrightarrow{22}$ , and  $\overrightarrow{23}$  to  $E_{cov}$ , and edge-pairs  $\overrightarrow{012}$ ,  $\overrightarrow{122}$ ,  $\overrightarrow{222}$ , and  $\overrightarrow{223}$  to  $EP_{cov}$ . As a result, the coverage levels for the regular expression  $\backslash d+$  by input strings  $S = \{s_0, s_1\}$  are:  $NC = 80\% (4/5)$ ,  $EC = 71.4\% (5/7)$ , and  $EP = 62.5\% (5/8)$ .

As an example of a non-matching string, let  $s_2 = "u"$ , which is interpreted as the byte stream [117 256]. The path traversed in  $G$  is  $0 \rightarrow e$ ; after reaching  $e$ , the processing stops. Node  $e$  is added to  $N_{cov}$ , edge  $\overrightarrow{0e}$  is added to  $E_{cov}$ , and there is no change to  $EP_{cov}$ . Considering  $S = \{s_0, s_1, s_2\}$ , the combined coverage levels are:  $NC = 100\% (5/5)$ ,  $EC = 85.7\% (6/7)$ , and  $EPC = 62.5\% (5/8)$ .

For another example of a non-matching string, let  $s_3 = "100u"$ , which is interpreted as the byte stream [49 48 48 117 256]. The path traversed in  $G$  is  $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow e$ . While this input visits all nodes in  $G$ ,  $NC = 100\%$  already, so no nodes are added to  $N_{cov}$ . Edge  $\overrightarrow{3e}$  is added to  $E_{cov}$ , edge-pair  $\overrightarrow{23e}$  is added to  $EP_{cov}$ . Considering  $S = \{s_0, s_1, s_2, s_3\}$ , the combined coverage levels are:  $NC = 100\% (5/5)$ ,  $EC = 100\% (7/7)$ , and  $EPC = 75\% (6/8)$ .

For each coverage metric, we compute coverage over the entire set of input strings, *total*, and two subsets: *success*, and *failure*. The numbers reported in this section are for the *total* set of input strings, that is,  $S = \{s_0, s_1, s_2, s_3\}$ . After, we split the input strings into those that terminate in an accept state in  $N_m$ , which we call  $S_{succ}$ , and those that terminate in the error state  $N_e$ , which we call  $S_{fail}$ . With this example,  $S_{succ} = \{s_0, s_1\}$  and  $S_{fail} = \{s_2, s_3\}$ .

Table 4.1 presents a summary of the coverage levels for each set of input strings. Achieving 100% for any of the coverage metrics is infeasible for  $S_{succ}$  alone because the error state  $e$  will never be reached, missing that node and the edges leading to it. In this example, EC for  $S_{succ}$  is 71.4% while EC for  $S$  is 100%.

Achieving 100% coverage for *EPC* is the most difficult, but it is possible in this example. The missing edge-pairs are computed by  $EP \setminus EP_{cov} = \{\overrightarrow{123}, \overrightarrow{13e}\}$ . Two additional input strings can lead to 100% EPC. Input “1u” would be interpreted as the byte stream [49 117 256] and traverses the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow e$ , hence covering  $\overrightarrow{13e}$ . Input “11u” would lead to byte stream [49 49 117 256], traverse the path  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow e$  and cover  $\overrightarrow{123}$ .

Note that it is possible to have a DFA which is simply two nodes connected by a single edge. Thus, edge pairs may not exist. For this case, we treat edge-pair coverage as identical to edge coverage.

## 4.4 Research Questions

To explore the potential of using graph coverage metrics for regular expressions, we evaluate the following research questions:

**RQ1:** *How well are regular expressions tested in GitHub?*

To answer RQ1, we identify 1,225 Java projects that have existing test suites covering the regular expressions. From these, we extract 15,096 regular expressions and 899,804 total test input strings, measuring NC, EC, and EPC for each regular expression. To obtain the regular expressions and their corresponding strings which are covered by test cases, we use the Java bytecode manipulation framework Javassist [35] to record the regular expressions when pattern matching methods are triggered by test cases.

**RQ2:** *How well can the regular expression string generation tool Rex improve the test coverage of regular expressions?*

Using the regular expressions from RQ1, we generate test strings using Rex [195] and

calculate the regular expression coverage, comparing it to the coverage of the user-defined test suites from RQ1. Using Rex, we generate test suites of three sizes, one to match the size of the user-defined test suites from the GitHub projects, one 5x that size, and one 10x that size. By comparing the coverage statistics we got in RQ2 to those in RQ1, we evaluate the test coverage possibilities through using an automated tool.

## 4.5 Study

Applying the coverage metrics defined in Section 4.3.2 to regular expressions from the wild requires 1) instrumentation to capture the regular expressions and strings matched against them (Section 4.5.1), 2) a tool to measure coverage given a regular expression and a set of strings (Section 4.5.2), and 3) a large corpus of projects with regular expressions and test suites that execute the regular expressions (Section 4.5.3). To address RQ2, we use the Rex [195] tool to generate input strings for the regular expressions in our study (Section 4.5.4).

### 4.5.1 Instrumentation

This section describes our approach to collecting regular expressions from GitHub projects and the strings evaluated against the regular expressions during testing.

#### 4.5.1.1 Instrumented Functions

There are different types of matching between a regular expression and a string. The Java function *Pattern.matches* requires the regular expression to match a string from its beginning to its end; Python's *re.match* requires the regular expression to match a string only from its beginning, not necessarily match to the end of the string; and the C# function *Regex.Match* requires the regular expression to match only a substring of the input string. These are called *FullMatch*, *FirstMatch*, and *ManyMatch*, respectively. In this chapter, we consider only *FullMatch* matches and related functions in Java projects. The related functions for FullMatch in Java are:

- `java.lang.String.matches(String regex)`
- `java.util.regex.Matcher.matches()`
- `java.util.regex.Pattern.matches(String regex, CharSequence input)`

In these functions the entire string is required to match the regular expression [60]. Thus, a regular expression with end-point anchors (i.e., `^` and `$`) and without are no different.

#### 4.5.1.2 Bytecode Manipulation

Our instrumentation is built on top of the Java bytecode manipulation framework `Javassist` [35], which can dynamically change the class bytecode in the JVM. All the projects are run in `jdk1.7`. We intercepted `FullMatch` function invocations in Java. For each invocation, we collect information about the regular expression itself, its location in the code, and any strings matched against it during test suite execution. These strings matched against the regular expression are referred to as the *input strings* or *test inputs* (i.e.,  $S$  from Section 4.3.2).

Since a regular expression may also appear in third-party libraries, we use the Java Reflection API to additionally record the caller function stack of the instrumented methods and extract the file name, class name, and method name of their caller methods. This allows us to identify when the regular expression being executed is from the system under test and when it is from a third-party library. We are dependent on two libraries during the experimentation, `org.junit` and `org.apache.maven`. Because Maven uses regular expressions to automate unit tests, all recorded regular expressions whose test classes are from package `org.junit.runner.*` or from package `org.apache.maven.plugins.*` are treated as regular expressions from third-party libraries and dropped.

#### 4.5.1.3 Recorded Information

We illustrate the recorded information for the regular expression `((:\w+)|\*)` and a string “one-name” from a project used in our study:<sup>4</sup>

- system under test: `mikko-apo/KiRouter.java`
- test file: `SinatraRouteParser.java`
- test class: `kirouter.SinatraRouteParser`
- test method: `compileRoutePattern`
- call site: line 38
- regular expression: `((:\w+)|\*)`

---

<sup>4</sup><https://github.com/mikko-apo/KiRouter.java>

- input string: "one-name"

In Section 4.2.3, the regular expression in the *call site* on line 1 is hard-coded. However, often the regular expression is passed as a variable, allowing multiple regular expressions to be observed during testing at the same call site (i.e., there is a many-to-one relationship between regular expressions and call sites). When this occurs, the recorded information is the same as above, except *regular expression* and *input string* would be different.

## 4.5.2 Coverage Analysis

This section details the construction of DFAs for computing coverage. Given a regular expression  $R$  and a set of input strings  $S$ , we first build a DFA for  $L(R)$  and then track the nodes and edges visited in the DFA during pattern matching with each string  $s \in S$ . We built our infrastructure on top of RE2 [42], a regular expression engine similar to those used in PCRE, Perl, and other languages.<sup>5</sup>

### 4.5.2.1 DFA Types.

Given a regular expression and an input string to match, we could build multiple DFAs with different considerations. We could build a static DFA with a regular expression alone or build a DFA on-the-fly (dynamic DFA) considering both a regular expression and an input string. For the same regular expression, different input strings will yield different dynamic DFAs. We can also build a Forward DFA and Backward DFA depending on the direction of scanning the regular expression. These decisions come with various performance tradeoffs during the matching process. For the purpose of our work, we need each DFA to be built consistently regardless of the input string, so we use a static DFA. We chose the forward direction as it seems the most natural for interpretation.

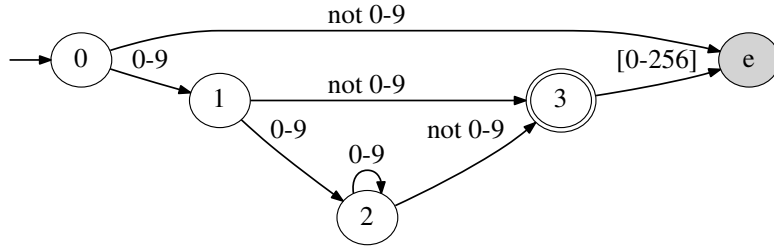
### 4.5.2.2 DFA Mapping

When matching an input string to a regular expression, RE2 builds a dynamic DFA. However, our coverage is computed over a static DFA. This requires mapping to aggregate coverage of a regular expression given multiple input strings.

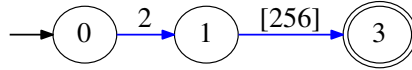
For a single regular expression, different input strings often result in different dynamic DFAs. To make matters worse, these DFAs have inconsistent naming of their states.

---

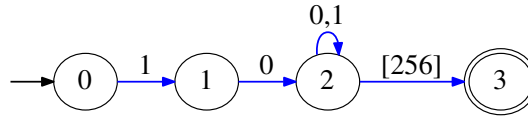
<sup>5</sup>Original RE2 at <https://github.com/google/re2> and modified code at <https://github.com/wangpeipei90/re2>.



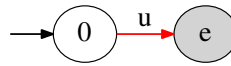
(a) Fully specified static DFA for:  $\backslash d^+$



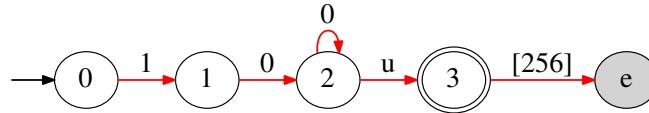
(b) Dynamic DFA for regular expression:  $\backslash d^+$  and input: "2"



(c) Dynamic DFA for regular expression:  $\backslash d^+$  and input: "1001"



(d) Dynamic DFA for regular expression:  $\backslash d^+$  and input: "u"



(e) Dynamic DFA for regular expression:  $\backslash d^+$  and input: "100u"

**Figure 4.6** Visited DFA subgraphs for the regular expression ' $\backslash d^+$ '. For each figure,  $N_0$  is the initial node 0,  $N_m$  is the accept node 3,  $N_e$  is the error node e. The arrows colored blue represent transitions in successful matches. The arrows colored red represent transitions in failed matches. The characters without square brackets are the literal characters in state transitions. For example, 'u' prompts the transition from Node 0 to Node e. [256] implies that there are no more bytes from the input string.

Therefore, to calculate the coverage of a certain regular expression based on the same DFA, these dynamic DFAs have to be mapped to the same static DFA, and then coverage is computed on the static DFA. This is usually straightforward as the dynamic DFA is always an isomorphic subgraph of the static DFA and  $N_0$ ,  $N_e$  and  $N_m$  are consistently labeled in the static and dynamic DFAs.

Consider the regular expression  $\backslash d^+$  and  $S = \{s_0, s_1, s_2, s_3\}$  from Section 4.3.2 where

$s_0 = \text{"2"}$ ,  $s_1 = \text{"1001"}$ ,  $s_2 = \text{"u"}$ , and  $s_3 = \text{"100u"}$ . Figure 4.6a shows the static forward DFA. The dynamic DFAs corresponding to these four inputs are shown in Figure 4.6b, Figure 4.6c, Figure 4.6d, and Figure 4.6e, respectively. Blue arrows are used to identify the visited edges in the dynamic DFAs when the input string is a match. Red edges are used to identify the visited edges when the input string is not a match. Note that in Figure 4.6, for simplicity, we have already mapped and renamed the nodes in the dynamic DFAs according to the static DFA.

### 4.5.2.3 RE2 Limitations and Modifications

We enlarged the default memory size of a cached DFA so that it could accommodate large DFA graphs. Due to Linux environment limitations, string length is limited to 131,072 and null type is not allowed. These situations are rare, impacting  $< 1\%$  of the collected regular expressions (see Section 4.5.3).

### 4.5.2.4 Coverage Calculation

With the consistent naming between a static DFA and a dynamic DFA, all nodes, edges, and edge pairs in the latter are regarded as visited nodes, edges, and edge pairs of the former. That is, a node only appears in a dynamic DFA when it is visited during matching; these can be thought of as *just-in-time* DFA constructions in the context of a string to match. The coverage metrics from Section 4.3.2 are computed over the static DFAs, aggregating over all input strings observed during testing.

## 4.5.3 Artifacts for RQ1

RepoReaper [132] provides a curated list of GitHub projects with the ability to sort based on project properties, such as the availability of test suites, which is a pre-requisite for our study. We focused on Java projects due to its popularity on GitHub and the availability of a bytecode analysis framework for instrumentation.

### 4.5.3.1 Project Selection

In December 2017, we selected the 136,196 Java projects whose unit test ratio reported in RepoReaper is greater than zero. Because the density of regular expressions in projects tends to be low, we automated project builds and test suite execution in order to collect sufficient data. As such, we require all projects we analyze to use *maven* and *junit* to

automatically run unit tests. We identified 13,637 Java Maven projects that used Java pattern matchings functions mentioned in Section 4.5.1. From those, we selected the ones that could be successfully compiled and tested in Maven, leaving 5,691 projects on which we attempted to collect coverage information.

#### 4.5.3.2 Regular Expression and Test Input Collection

To collect the input strings for each regular expression, we instrumented each project and executed the test suites. We changed the configurations of the plugin *maven-surefire-plugin* by adding *-javaagent* argument to *argLine* so that when Maven forks a VM to run the unit tests the VM can load the instrumentation library. Each project module that runs tests executes in different VMs and the information is recorded in different files. *testFailureIgnore* is configured to *true* so that one test failure does not affect the other tests, allowing us to record as many regular expressions in the project as possible.

Of the 5,691 projects with Maven, test suites, and pattern matching functions, 1,665 projects contained 24,058 regular expressions executed by test suites. The remaining projects contained regular expressions *not* executed by the test suites, and thus could not be instrumented.

#### 4.5.3.3 Filtering Out Third-Party Regular Expressions

FullMatch invocations from Maven and JUnit have been removed already at this point, but other third-party libraries also use regular expressions. We can detect this by looking for syntactically identical regular expressions with invocations on the same file, same class, same method, but in different GitHub projects. If the number of projects is larger than one, then it is regarded as a third-party regular expression, and all records related to the same stack information are dropped. A limitation of this approach is that we miss some third-party invocations that are only present in a single project. Given the large number of projects analyzed, the impact of this is likely to be small.

We identified 8,496 regular expressions as coming from third-party libraries. The resulting dataset contains 1,256 projects and 15,562 regular expressions, 14,040 of which are syntactically unique.

#### 4.5.3.4 RE2 Analysis

Since RE2 only supports the most common regular expression language features, we filtered out the regular expressions containing advanced and non-regular features. RE2

**Table 4.2** Description of 1,225 Java projects analyzed. All numbers are rounded to nearest integer except the test ratio and KLOC.

Attributes	mean	25%	50%	75%	90%	99%
Tested Regular exp.	12	1	3	7	18	99
Stars	35	0	1	5	30	833
Test ratio	0.238	0.096	0.210	0.346	0.482	0.691
KLOC	55.4	2.0	6.7	25.1	86.7	951.0
Size (KB)	19,062	286	1,079	6,449	33,163	249,915
Call sites	15	2	4	10	31	211
Tested call sites	3	1	2	3	6	20
Reg. exp./tested site	5	1	1	2	5	38

**Table 4.3** Description of 15,096 regular expressions analyzed for RQ1. All numbers are rounded to nearest integer.

Attributes	mean	25%	50%	75%	90%	99%
Nodes ( $ N $ )	144	12	28	70	324	939
Edges ( $ E $ )	565	24	75	212	938	2,813
Edge pairs ( $ EP $ )	2,115	25	99	414	1,647	16,850
Regular exp. len.	31	13	18	39	67	161
# Input strings ( $ S $ )	60	1	2	7	27	662
Input string len.	125	9	17	63	318	948

failed to construct DFAs for 457 regular expressions, leaving 15,105 regular expressions spread across 1,225 projects.<sup>6</sup> The RE2 limitations on input string length and the null byte affected 56 regular expressions and 191 input strings, and nine of the 56 regular expressions are removed from coverage analysis because their only input string is dropped.

These 1,225 projects contain 18,426 call sites of the instrumented functions. Only 3,093 call sites are executed by the test suites; the same call site can have many regular expressions in the case of dynamically generated regular expressions.

The final dataset used for analysis contains 1,225 projects, 3,093 call sites, 15,096 regular expressions, of which 13,632 are syntactically unique. As the same regular expression can appear in multiple projects, or multiple places in the same project, all are retained since each is potentially tested differently. These 15,096 regular expressions are executed by 899,804 test inputs.

<sup>6</sup>Assuming all 457 are non-regular, this means over 97% of the regular expressions sampled are regular, echoing findings from the Python analysis in Section 4.2.2.

### 4.5.3.5 Project Characteristics

Table 4.2 describes the 1,225 projects in terms of *Tested regular exp.* (numbers of tested regular expressions per project), *stars* (a measure of popularity), *KLOC* (lines of code in thousand), *size* (size of the repository in KB), *test ratio* (the ratio of number of lines of code in test files to the total lines of code in repository, as reported by RepoReaper), *Call sites* (the number of FullMatch methods in the source code), *Tested call sites* (the number of FullMatch call sites executed by the tests), and *Reg. exp. / tested site* (the number of regular expressions passed to each tested call site). The *mean* column describes the average value for each attribute. Columns *25%*, *50%*, *75%*, *90%*, and *99%* show the distribution of each attribute at 25 percentile, median, 75 percentile, 90 percentile, and 99 percentile, respectively. The average number of tested regular expressions collected per project was 12 with a range of 1 to 2,004.

### 4.5.3.6 Regular Expression Characteristics

Table 4.3 shows the DFA information for regular expressions. *Nodes*, *edges*, and *edge pairs* are the total number of nodes, edges, edge pairs in the DFA graph of a regular expression. The average regular expression is quite large with 144 nodes, though this is skewed as the median is 28 nodes. *Regular exp. len.* measures the length of the string representing the regular expression itself in characters. *# Input strings* is the number of syntactically unique input strings executed by a project’s test suite, per regular expression. The average number of syntactically unique test inputs per regular expression is 60, but the median is 2. *Input string len.* shows the lengths of the input strings (i.e., each  $s \in S$ ) in terms of the number of characters.

## 4.5.4 Artifacts for RQ2

To explore the coverage of regular expressions using tools, we selected Rex [195] due to its high language feature coverage [31].

### 4.5.4.1 Artifact Selection

We need a set of regular expressions with the following characteristics: 1) are covered by tests; 2) can be analyzed by RE2 for coverage analysis; and 3) can be analyzed by Rex for test input generation. To satisfy 1) and 2), we begin with the dataset from RQ1 of 1,225 projects and 15,096 regular expressions. To satisfy 3), we select all the regular expressions

**Table 4.4** Description of 7,926 regular expressions for RQ2.

Attributes	mean	25%	50%	75%	90%	99%
Nodes ( $ N $ )	220	13	31	162	618	970
Edges ( $ E $ )	773	30	97	663	1,468	3,694
Edge pairs	2,422	36	186	1,021	1,999	21,274
$ S $	70	1	2	8	39	961
$ S_{succ} $	34	1	1	2	8	208
Regular exp. len.	29	12	15	31	71	160

that Rex supports and for which  $|S_{succ}| > 0$ , since Rex only generates matching strings, leaving 10,155 regular expressions of which 9,063 are syntactically unique.

#### 4.5.4.2 Rex Setup

Rex defaults to *ManyMatch* as opposed to the *FullMatch* behavior of our dataset. To force Rex to treat each regular expression as a full match, we added endpoint anchors (i.e.,  $\wedge$  and  $\$$ ) to each regular expression. Because Rex may get stuck in generating input strings for certain regular expressions, we set a timeout of one hour for Rex to generate strings; regular expressions that exceed the timeout are discarded. Of the 10,155 regular expressions in GitHub whose  $S_{succ} > 1$ , Rex encountered the timeout for only two.

Another complication comes at the intersection of the Rex and RE2 language support; Rex-generated strings must be processed by RE2 for the coverage analysis. For example, the character class “ $\backslash s$ ” in Rex accepts six whitespace characters and RE2 accepts five. In another example, some generated Unicode strings in Rex could not be processed in RE2 because their Unicode encoding in Rex is UTF-16 while RE2 handles Unicode sequences encoded in UTF-8 or Latin-1. To simplify the experiment, we configured Rex to generate strings in ASCII. We also dropped strings which contain unsupported features or characters in either RE2 or Python 3. We also dropped strings which lead to failed matchings and reported the coverage based on successful matchings.

After filtering out all the unsupported regular expressions, our reported coverages by Rex strings in ASCII encoding are based on 7,926 regular expressions of 985 GitHub projects; 7,007 of them are syntactically unique. Table 4.4 shows the attributes of regular expressions of which Rex could generate regular expressions.

### 4.5.4.3 Input String Generation

For each regular expression  $R$ , we use Rex to generate input string sets relative to the size of the matching strings  $|S_{succ}|$ . We generate input string sets of three sizes: equal to  $|S_{succ}|$ ; equal to  $5 \times |S_{succ}|$ ; and equal to  $10 \times |S_{succ}|$ . We refer to these experiments as *Rex1M*, *Rex5M*, and *Rex10M*, respectively. For each experiment, we repeated the string generation using the system time as the random seed to encourage diversity among the generated strings. The averages over five runs (*Rex5M* and *Rex10M*) or ten runs (*Rex1M*) for each metric are reported as Rex’s coverage of  $R$ .

For example, say a regular expression  $R$  from GitHub has five input strings;  $|S| = 5$ . Three of the input strings are matching;  $|S_{succ}| = 3$ . For this experiment, Rex would generate three strings ten times, then 15 strings five times, then 30 strings five times, totaling  $30 + 75 + 150 = 255$  generated strings. For each set of  $\{3, 15, 30\}$  strings, NC, EC, and EPC are computed, averaged over  $\{10, 5, 5\}$  runs.

In the case of finite languages, Rex may fail to generate sufficient input strings. For example, the total number of matching input strings in ASCII for a regular expression  $\backslash d$  is ten (i.e., 0-9). If in the repository there are also three matching input strings, Rex could generate three strings ten times, but would fail to generate  $5 \times 3 = 15$  strings. The calculation of NC, EC, and EPC are based on the best-effort: for each run of every regular expression, we calculate coverage with input strings up to  $|S_{succ}|$  in *Rex1M*, 5x of  $|S_{succ}|$  in *Rex5M*, and 10x of  $|S_{succ}|$  in *Rex10M*; and coverage of every regular expression is the averages of its coverages over  $\{10, 5, 5\}$  runs in *Rex1M*, *Rex5M*, and *Rex10M*. In other words, if Rex failed to generate required number of input strings, the coverage is calculated based on the input strings Rex can generate.

In the ten runs of generating input string sets equal to  $|S_{succ}|$  for *Rex1M*, there are 833 regular expressions which have input strings less than  $|S_{succ}|$  in at least one run. In the five runs of generating input string sets 5x of  $|S_{succ}|$  for *Rex5M*, there are 2,041 regular expressions which have input strings less than 5x of  $|S_{succ}|$  in at least one run. In the five runs of generating input string sets 10x of  $|S_{succ}|$  for *Rex10M*, there are 2,336 regular expressions which have input strings less than 10x of  $|S_{succ}|$  in at least one run.

## 4.6 Results

Here, we present the results of RQ1 and RQ2 in turn.

**Table 4.5** Description of 15,096 Regular Expressions’ test suites. All numbers are rounded to nearest integer except that success ratio rounded to two decimal places.

Attributes	mean	25%	50%	75%	90%	99%
$ S $	60	1	2	7	27	662
$ S_{succ} $	19	0	1	1	4	79
$ S_{fail} $	41	0	1	4	19	383
succ_ratio	49.03	0.00	44.70	100.00	100.00	100.00
fail_ratio	50.97	0.00	55.30	100.00	100.00	100.00

## 4.6.1 RQ1: Test Coverage of Regular Expressions

We address RQ1 in two ways. First, we look at the number of call sites to FullMatch methods that are actually tested. Next, we look at the test coverage for each tested regular expression.

### 4.6.1.1 Tested Call Sites

In the 1,225 projects, there are 18,426 call sites of the instrumented functions in Section 4.5.1.1. However, only 3,093 call sites are executed by the test suites. This means that 15,333 (83.21%) of the call sites are not covered by the test suites. For those that are, the median of unique regular expressions per tested call site is one, with an average of five (Table 4.2).

**Summary:** Of the 18,426 call sites for FullMatch methods in 1,225 GitHub projects, only 3,093 (16.8%) are executed by the test suites.

### 4.6.1.2 Coverage of Tested Regular Expressions

We successfully generated static DFAs for 15,096 regular expressions from 1,225 Java GitHub projects and dynamic DFAs for 899,804 regular expression/input string pairs.<sup>7</sup> Among the regular expressions, 4,941 (32.7%) regular expressions have only failing inputs (i.e.,  $|S_{succ}| = 0$ ) and 6,029 (39.9%) have only inputs of successful matching (i.e.,  $|S_{fail}| = 0$ ). This means that 10,970 (72.7%) of the regular expressions do not contain test inputs that exercise both the matching and non-matching scenarios. Of these, 6,318 (41.9%) regular

<sup>7</sup>We note that 899,804 is less than  $60 \times 15096 = 905760$  because the mean of  $\# \text{Input strings } (|S|)$  is 59.60546 and rounded up to 60.

**Table 4.6** Coverage values in Figure 4.7.

Coverage	Suite	mean	25%	50%	75%	90%	99%
NC (%)	$S$	59.05	24.62	63.64	95.65	100.00	100.00
NC (%)	$S_{succ}$	47.84	0.00	46.15	90.00	99.60	99.89
NC (%)	$S_{fail}$	18.89	0.00	8.51	25.00	62.26	100.00
EC (%)	$S$	28.74	6.67	23.90	49.97	53.80	80.00
EC (%)	$S_{succ}$	23.20	0.00	12.36	49.96	50.00	60.00
EC (%)	$S_{fail}$	8.55	0.00	2.20	7.80	32.19	65.08
EPC (%)	$S$	23.77	2.47	12.50	49.96	50.00	66.67
EPC (%)	$S_{succ}$	20.48	0.00	5.26	49.94	50.00	55.56
EPC (%)	$S_{fail}$	5.50	0.00	0.00	2.74	22.12	57.14

expressions contain only one test string (i.e.,  $|S| = 1$ ) There are 4,126 (27.3%) regular expressions with both failed and successful matchings.

Table 4.5 describes properties of the test input sets for each regular expression:  $|S|$  is the size of the test suite, computed as the number of unique input strings for a regular expression;  $|S_{succ}|$  means the number of matching inputs;  $|S_{fail}|$  means the number of failing inputs; *succ\_ratio* shows the ratio of successful matchings to all matchings for each regular expression; *fail\_ratio* shows the ratio of failed matchings to all matchings for each regular expression.

Table 4.6 describes the distributions of Node Coverage (NC), Edge Coverage (EC), and Edge-Pair Coverage (EPC) over  $S$ ,  $S_{succ}$ , and  $S_{fail}$ . Figure 4.7 displays this information graphically; *total* means the total number of input strings for each regular expression, namely the test suite  $S$ , *success* means  $S_{succ}$ , and *failure* means  $S_{fail}$ . Most of the regular expressions are not tested thoroughly since the mean values of coverage are low, especially the edge and edge-pair coverage. Although the coverages on failed matchings are relatively small, they contribute to a high overall test coverage. Failed matching tests are a necessary part of testing regular expressions.

**Summary:** A majority of regular expressions (10,970, 97.7%) are tested with exclusively passing (6,029, 39.9%) or exclusively failing (4,931, 32.7%) test inputs. Edge and edge-pair coverage are both very low. Full node coverage is infeasible with only passing inputs as  $N_e$  would never be covered; the presence of both types of inputs is important for thorough testing.

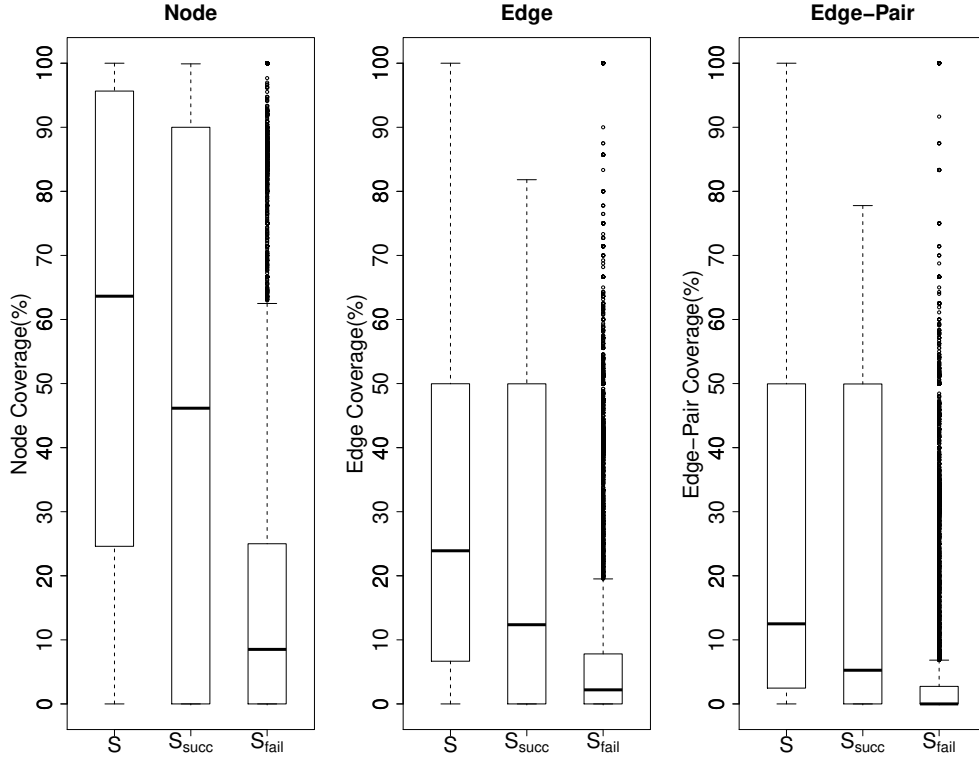
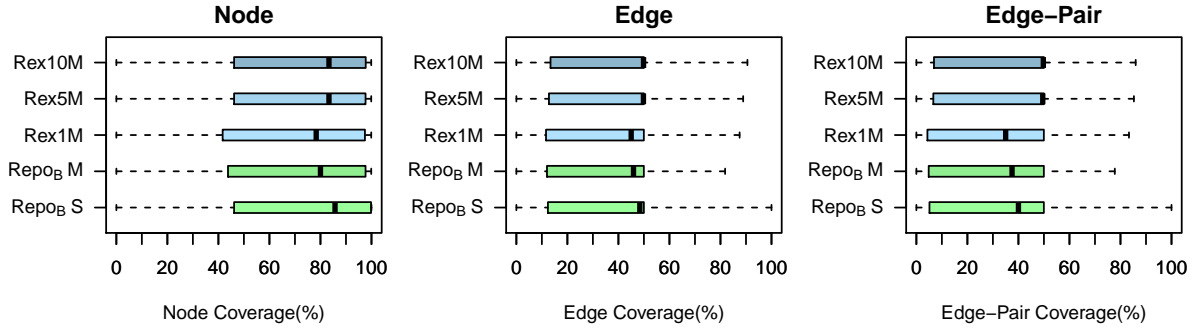


Figure 4.7 Coverage for 15,096 regular expressions.

## 4.6.2 RQ2: Coverage with Rex

Figure 4.8 shows the analysis results given the generated inputs in ASCII encoding, organized by each of five datasets.  $Repo_B S$  and  $Repo_B M$  show the coverages over  $S$  and  $S_{succ}$  of 7,926 regular expressions using the developer-defined test suite in GitHub and their details are in Table 4.7.  $Rex1M$ ,  $Rex5M$ , and  $Rex10M$  show the coverages of 7,926 regular expressions based on the Rex-generates test inputs with sizes of 1x, 5x, and 10x of the user-defined test suite, respectively. Coverage details are shown in Table 4.8.

Table 4.9 illustrates the differences in coverage between the repository ( $Repo_B M$  and  $Repo_B S$ ) and Rex ( $Rex1M$ ,  $Rex5M$ , and  $Rex10M$ ). Using a paired Wilcoxon signed-rank test, we find that for all three coverage metrics,  $Repo_B M$  significantly outperforms  $Rex1M$  with  $\alpha = 0.0001$ . However, as test suite size is strongly correlated with coverage [83], as soon as the Rex test set is amplified to 5x and 10x the size, the coverage of Rex outperforms the repository. When considering all test inputs from the repository and not just the successful ones, with test inputs sets of the same size,  $Repo_B S$  outperforms  $Rex1M$ . However, this comparison is unfair since Rex does not generate



**Figure 4.8** Node, edge, edge-pair coverage of 7,926 regular expressions with Rex-generated ASCII inputs (*Rex1M*, *Rex5M*, *Rex10M*) of 7,926 regular expressions in GitHub which are used in Rex (*Repo<sub>B</sub>M*, *Repo<sub>B</sub>S*).

**Table 4.7** Coverage values of the 7,926 regular expressions in GitHub for *Repo<sub>B</sub>M* and *Repo<sub>B</sub>S* in Figure 4.8.

Coverage	Expr	mean	25%	50%	75%	90%	99%
NC (%)	<i>Repo<sub>B</sub>M</i>	70.41	43.75	80.00	97.67	99.84	99.90
EC (%)	<i>Repo<sub>B</sub>M</i>	33.79	12.01	45.91	49.97	50.00	66.67
EPC (%)	<i>Repo<sub>B</sub>M</i>	29.39	4.83	37.50	49.97	50.00	60.00
NC (%)	<i>Repo<sub>B</sub>S</i>	73.27	46.15	85.71	99.83	100.00	100.00
EC (%)	<i>Repo<sub>B</sub>S</i>	36.35	12.36	48.39	49.97	60.00	85.71
EPC (%)	<i>Repo<sub>B</sub>S</i>	30.68	5.13	40.00	49.97	50.00	74.67

**Table 4.8** Coverage values of the 7,926 regular expressions using Rex for *Rex1M*, *Rex5M*, and *Rex10M* in Figure 4.8.

Coverage	Expr	mean	25%	50%	75%	90%	99%
NC (%)	<i>Rex1M</i>	69.29	41.67	78.33	97.44	99.84	99.90
EC (%)	<i>Rex1M</i>	33.57	11.62	45.00	49.97	50.00	71.43
EPC (%)	<i>Rex1M</i>	29.50	4.33	35.00	49.96	50.00	66.67
NC (%)	<i>Rex5M</i>	71.69	46.15	83.33	97.67	99.84	99.90
EC (%)	<i>Rex5M</i>	36.42	12.77	49.81	50.00	54.55	80.00
EPC (%)	<i>Rex5M</i>	33.04	6.63	49.54	50.00	56.67	75.00
NC (%)	<i>Rex10M</i>	72.01	46.15	83.33	97.73	99.84	99.90
EC (%)	<i>Rex10M</i>	36.87	13.39	49.85	50.00	55.89	80.00
EPC (%)	<i>Rex10M</i>	33.77	6.90	49.77	50.00	58.33	75.00

non-matching strings. That said, as soon as the Rex dataset is amplified as in *Rex5M* and *Rex10M*, there is no clear winner compared to all test inputs from the repository. While

**Table 4.9** Differences in coverage based on datasets in Figure 4.8. Hypothesis tests used paired Wilcoxon signed-rank test. Bold text identifies when one of the datasets had significantly higher coverage for all three metrics. If there was a conflict between the metrics (e.g., Set1 > Set2 for NC, and Set1 < Set2 for EPC), there was no winner.

Set1	Set2	$H_0 : Set1 \stackrel{d}{=} Set2$		
		NC	EC	EPC
<b>Repo<sub>B</sub>M</b>	<i>Rex1M</i>	p < 0.0001	p < 0.0001	p < 0.0001
<i>Repo<sub>B</sub>M</i>	<b>Rex5M</b>	p < 0.0001	p < 0.0001	p < 0.0001
<i>Repo<sub>B</sub>M</i>	<b>Rex10M</b>	p < 0.0001	p < 0.0001	p < 0.0001
<b>Repo<sub>B</sub>S</b>	<i>Rex1M</i>	p < 0.0001	p < 0.0001	p < 0.0001
<i>Repo<sub>B</sub>S</i>	<i>Rex5M</i>	p < 0.0001	p = 0.0004	p < 0.0001
<i>Repo<sub>B</sub>S</i>	<i>Rex10M</i>	p < 0.0001	p = 0.4147	p < 0.0001
<b>Repo<sub>B</sub>S</b>	<i>Repo<sub>B</sub>M</i>	p < 0.0001	p < 0.0001	p < 0.0001

it may appear that Rex can do as well as the repository, the reality is that the error node will never be covered by Rex, a fact which is not apparent by looking at the numbers alone.

**Summary:** Rex can handle approximately 78.1% of the regular expressions from our dataset. Considering only the matching test inputs and test sets of the same size, Rex does not achieve coverage as high as the developer-written tests. However, the coverage numbers are extremely close. This indicates that tools such as Rex can be used to write test inputs with similar coverage to the developer tests, but will always miss  $N_e$  and all edges incident to it.

## 4.7 Threats to Validity

**Internal:** We measure the test coverage of regular expression used in functions of full matching with FullMatch DFAs in the forward direction. The experimental results may not reflect the test coverage of regular expressions used in other functions, nor the test coverage of regular expressions which could not be converted into a DFA.

**External:** The Java regular expressions used in this evaluation were collected from RepoReaper Java Maven projects compiled with Java jdk1.7, which is only a small portion of all GitHub Java projects and may not generalize to all Java projects and to other languages. It is possible that there are still regular expressions from third-party libraries in the dataset, which could bias results. Due to limitations of RE2 and Rex, the results of test coverage applies exclusively to the features supported. All our projects had test

suites, which may overestimate the test coverage levels for typical regular expressions.

## 4.8 Summary

To our knowledge, this is the first research to evaluate fine-grained coverage metrics for regular expressions. We explore coverage over the DFA representation of a regular expression. It is also the first research to explore how often regular expressions are tested in a large set of the software projects. We show the coverage metrics of regular expressions from 1,225 GitHub Java Maven projects and found that over 80% of *FullMatch* functions are not tested and that most of the tested regular expressions have a low edge and edge-pair coverage. We also show that with the help of the regular expression tool Rex it is possible to improve the regular expression testing coverage by adding input strings, but that there is an upper bound for this type of improvement. This research is a first step toward better understanding how regular expressions are tested in practice; future work will explore how various coverage metrics can reduce the bugs associated with regular expressions.

## CHAPTER

# 5

# REGULAR EXPRESSION EVOLUTION

Chapter 5 presents how regular expression evolves over time by studying the regular expression edits in terms of language features, semantic and syntactic similarities. Portions of this chapter were published in *SANER 2019* [198].

## 5.1 Introduction

In this chapter, we explore how regular expressions evolve over time. Beyond shedding light on the types of edits to consider in fault-based test generation, the history of source code development and the information of bug fixes are valuable in guiding program repairs [103]. Given the fault-proneness of regular expressions [141], and that developers under-test their regular expressions [199], understanding how they evolve can help guide testing and repair efforts. For example, if regular expressions typically increase in scope over time (i.e., the language expands), it is valuable to focus testing efforts on strings beyond a regular expression's language. If, however, regular expressions typically decrease in scope, testing efforts should focus within the regular expression's language.

This chapter presents a study of regular expression evolution on two datasets collected

separately from two different contexts: Github dataset and Video dataset. The GitHub dataset is comprised of Java regular expressions collected from GitHub projects by mining their source code commit history. It represents the use case of regular expressions in a persistent environment (e.g., within source code) and provides a coarse-grained view of changes to the regular expression over time. However, the commit history can mask the actual evolution of regular expressions as a developer is composing them since the commit history represents only what is pushed to the repository. The Video dataset contains Java regular expressions written by developers during problem-solving tasks. It was conducted in an ephemeral environment (e.g., grepping/searching a document/IDE) and used as a finer-granularity view into regular expression evolution. In combination, we are able to see the types of changes developers made to regular expressions in both contexts.

The syntactic measurement is Levenshtein distance [107] on evaluating how many characters are changed, and the semantic measurement is the semantic similarity between a regular expression’s language and its predecessor on evaluating how the matching strings changed. Our study shows that:

- 95% of literal regular expressions in GitHub (i.e., that do not contain variables) do not evolve (RQ1),
- Approximately half the edits in both datasets contain six or fewer character modifications (RQ2),
- Over half the edits in the GitHub dataset expand the scope of the language whereas the most common edits in the Video dataset create a disjoint language (RQ2), and
- Most edits in the GitHub and Video datasets involve adding and/or removing 4-6 language features (RQ3).

While most regular expressions do not change, understanding the common changes, and the changes to the language, provides valuable insights for designing mutation operators to assess the quality of test suites and for repairing faulty regular expressions, two promising directions of future work.

## 5.2 Motivation

While we show later that a majority of regular expressions do not evolve (see Section 5.6), prior work indicates that regular expression bug reports abound [179] and that regular

expressions are under-tested [199]. In exploring bug reports on regular expressions, we find evidence that regular expressions often evolve through refactoring (i.e., the language is the same), and that edits tend to involve many different mutation operators at once. This provides evidence that understanding evolution is important for test generation and repair.

### 5.2.1 Regular Expression Equivalence

A regular expression is a language to describe a set of strings it can match, and there is usually more than one way to express it. For example, a digit can be described as a character range `[0-9]` and can also be described using shortcut `\d`. A word character expressed in `\w` is equivalent to `[A-Za-z_0-9]`.<sup>1</sup> Sometimes, bug reports require resolution through semantics-preserving transformations that improve performance rather than edits to regular expression semantics. In one bug report [82], all regular expression capturing groups are changed to non-capturing groups (e.g., `(\r\n|\r|\n|\f)` to `(?:\r\n|\r|\n|\f)`) to avoid back tracking so the scope of the regular expression is not changed by the mutation. In another example [57], regular expression `(\W|\d|_)` is changed to `[^A-Za-z]` for better regular expression readability. (To clarify, `\W` is equivalent to the negated character class `[^A-Za-z_0-9]`, and `\d` is equivalent to `[0-9]`; making `[0-9]` valid in the desired character class; the underscore is treated similarly).

These provide evidence that not all edits modify the semantic, and thus not all testing efforts should focus on matching behavior [199], but rather on performance and understandability.

### 5.2.2 Regular Expression Feature Changes

The state-of-the-art literature on fault-injection [11] and fixing regular expressions [10] uses simple faults. However, fixing regular expressions in bug reports often involves more than one feature. For example [82], the regular expression

```
[+|-]?\\d*\\.?\\d+([a-z]+|%)?
```

is changed to

```
[+|-]?+(?:\\d++(?:\\.\\d++)?+|\\.\\d++)?(?:[a-z]++|%)?+
```

In this example, greedy quantifiers (e.g., `[+|-]?` and `\\d+`) are changed to possessive quantifiers (e.g., `[+|-]?+` and `\\d++`) seven times, and the capturing group is changed to

---

<sup>1</sup>More information about regular expression equivalence can be found in [31].

a non-capturing group.

Understanding the types and frequencies of edits in a regular expression’s evolution can shed light on how to guide fault-injection test generation and repair.

### 5.3 Research Questions

We explore regular expression evolution with respect to syntactic and semantic measures for the purpose of exploration. Here, we consider the regular expression string itself, referred to as  $r_1$  or  $r_2$ , and the languages described by the regular expressions,  $L(r_1)$  and  $L(r_2)$ . Evolution is explored in the context of two datasets: one from the commit histories of projects on GitHub, and one from screencasts of students solving regular expression tasks. Our research questions are:

**RQ1:** *What are the characteristics of regular expression evolution?*

We explore the number of edits as each regular expression evolves, the invalid regular expressions on edit chains, and the phenomenon of regular expression reversions, when the same regular expression re-appears later in an edit chain.

**RQ2:** *How similar is a regular expression to its predecessor syntactically and semantically?*

We explore syntactic similarity with the Levenshtein distance between regular expression strings over time. Regarding semantic similarity, we are interested in the languages described by regular expressions and how those change over time. For a regular expression  $r_2$  that evolves from  $r_1$ , we are interested in the overlap between the languages (i.e.,  $L(r_2)$  and  $L(r_1)$ ).

**RQ3:** *How do the features change in the evolution of a regular expression?*

We explore the features that are most frequently added, removed, or changed over the whole datasets. We also analyze the number of features added, removed or changed per regular expression edit.

### 5.4 Analysis

A series of edited regular expressions over time is called an *edit chain*, which represents the changing history of a regular expression. The *top* of the chain is the most recent version of the regular expression, whereas the *bottom* of the chain is the oldest version.

The *length* of the chain is the number of regular expressions in it, and the number of edits in the chain is *length - 1*. The bottom of the chain is  $r_1$  and the top is  $r_k$  for some length  $k$  of the chain. For  $r_i$  its *predecessor* is  $r_{i-1}$  and its *successor* is  $r_{i+1}$ . The similarity is described between *pairs* of adjacent regular expressions in the edit chain, referred to generically as an *edit*. More generally, we compare an edited regular expression  $r_i$  that evolved into its successor,  $r_{i+1}$ .

For a running example in this section, consider the regular expressions,  $r_1$  and  $r_2$ :

$r_1 = \text{caa?a?b}$

$r_2 = \text{c}\{0,2\}\text{aa?b}$

The languages of  $r_1$  and  $r_2$  are both finite, and enumerated below, with overlapping strings between  $L(r_1)$  and  $L(r_2)$  bolded for clarity:

$L(r_1) = \{\text{“cab”}, \text{“caab”}, \text{“caaab”}\}$

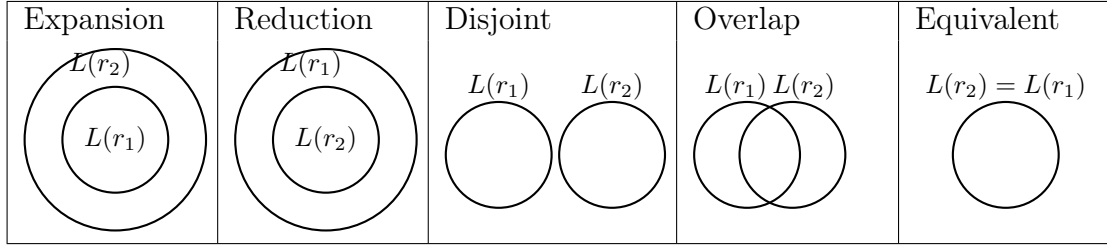
$L(r_2) = \{\text{“ab”}, \text{“aab”}, \text{“cab”}, \text{“caab”}, \text{“ccab”}, \text{“ccaab”}\}$

Considering the syntax between  $r_1$  and  $r_2$ , there are clear similarities and differences. Considering the overlapping semantics between languages, there are also clear similarities and differences. Here, we first describe how these similarities and differences are measured syntactically and semantically and then describe how feature changes are represented in the feature vector. In the end, we provide our implementation details and limitation discussion.

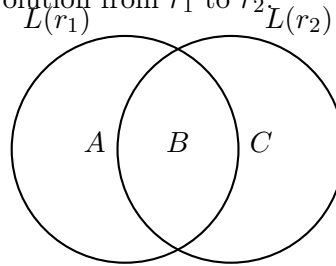
### 5.4.1 Syntactic Similarity

The syntactic similarity is measured by the distance between the literal representation of the strings. We followed the measurement of edit distance used in the work of the automatic generation of regular expressions with genetic programming [18] and thus chose Levenshtein distance.

Levenshtein distance [107] measures the syntactic distance between two strings by counting the number of character insertions, deletions, and substitutions needed to transform one regular expression into the other. In the example with  $r_1$  and  $r_2$ , the absolute Levenshtein distance between them is six. This is computed as one replacement (i.e.,  $a \rightarrow \{$ ), four additions (i.e., **0,2**) and one removals (i.e.,  $\}$ ) as follows: **ca****{0,2}****a?b**



**Figure 5.1** Types of semantic evolution from  $r_1$  to  $r_2$



**Figure 5.2** Notation for sets A, B and C to compute the semantic evolution from  $r_1$  to  $r_2$ .

## 5.4.2 Semantic Similarity

Semantic similarity measures the amount of overlap between  $L(r_1)$  and  $L(r_2)$ . Adopting the approach from the work of Chapman and Stolee [31], we measure approximate similarity by looking at the overlap in matching strings between two regular expressions. We define the evolution of  $r_1$  to  $r_2$  by measuring three sets of strings,  $A$ ,  $B$ , and  $C$ , which are represented in the Venn Diagram in Figure 5.2. Formally:

$$A = L(r_1) \setminus L(r_2),^2$$

$$B = L(r_1) \cap L(r_2), \text{ and}$$

$$C = L(r_2) \setminus L(r_1).$$

There are five types of relationships between  $L(r_1)$  and  $L(r_2)$ , which are shown in Figure 5.1.<sup>3</sup> When  $L(r_1)$  is a strict subset of  $L(r_2)$ , then more matching strings are added to the language; this is called *expansion*. When  $L(r_1)$  is a strict superset of  $L(r_2)$ , this means that the language was reduced during evolution; we call this *reduction*. If there is no overlap between  $L(r_1)$  and  $L(r_2)$ , then they are *disjoint*. If  $L(r_1)$  and  $L(r_2)$  are the same, they are called *equivalent*. The final condition is a *overlap* when  $L(r_2)$  and  $L(r_1)$  share some strings, but some are removed and some are added during evolution.

Using the example:

$$A = \{\text{“caaab”}\},$$

<sup>2</sup> $X \setminus Y = \{x \in X : x \notin Y\}$

<sup>3</sup>Prior work used mutation for test case generation and defined similar relationships between regular expressions, omitting disjoint [11].

$B = \{\text{“cab”}, \text{“caab”}\}$ , and  
 $C = \{\text{“ab”}, \text{“aab”}, \text{“ccab”}, \text{“ccaab”}\}$ .

Migrating from  $r_1$  to  $r_2$  involved an overlap, where all the strings in  $A$  are removed from the matching language, and all the strings in  $C$  are added. In this way,  $r_1$  and  $r_2$  are partially overlapped. We measure three metrics pertaining to the semantic evolution of regular expressions: *intersection*, *removal*, and *addition*.

#### 5.4.2.1 Intersection

The Intersection between  $L(r_1)$  and  $L(r_2)$  is computed as:

$$intersection(r_1, r_2) = \frac{|B|}{|A| + |B| + |C|}$$

When the languages are *disjoint*,  $|B| = 0$ , and so the intersection is likewise 0. When the languages are identical,  $A$  and  $C$  are empty, so the intersection is 1. In the running example with partially intersecting languages,  $\frac{2}{1 + 2 + 4} = \frac{4}{7} = 57\%$ .

#### 5.4.2.2 Removal

This metric describes how much of the language of  $r_1$  is removed in the migration to  $r_2$ . Generically, the reduction from  $r_1$  to  $r_2$  is:

$$removal(r_1, r_2) = \frac{|A|}{|A| + |B|}$$

When the  $r_2$  is an *expansion* of  $r_1$ , the removal is 0 since  $|A| = 0$ . When the languages are disjoint,  $B$  is empty, so the removal is 1. When the languages are equivalent,  $A$  is empty so the removal is 0. In our example of overlap, removal is  $\frac{1}{1 + 2} = \frac{1}{3} = 33\%$  meaning that 33% of the language of  $r_1$  was removed when evolving it to  $r_2$ .

We can compute the percentage of the  $L(r_1)$  that is retained in  $L(r_2)$  by  $1 - removal(r_1, r_2)$ . In this example, 67% of  $L(r_1)$  is carried forward into  $L(r_2)$ .

#### 5.4.2.3 Addition

This metric describes how much of the language  $L(r_2)$  is new or added after evolving from  $r_1$ . It is computed as:

$$addition(r_1, r_2) = \frac{|C|}{|B| + |C|}$$



provided in the Java library of Apache Commons Text.<sup>5</sup>

For semantic distance, we approximate the language for  $L(r)$  by generating strings for  $r$  using Rex [195]. The Rex tool analyzes regular expressions using symbolic analysis. When configured as a string generation tool, it aims to generate strings inside  $L(r)$ . Using Rex, for each regular expression  $r$ , we tried to generate  $k = 500$  strings which could successfully match  $r$ . Considering  $L(r)$  could be smaller than  $k$ , Rex tried up to five times for each  $r$  with different seeds so that the accumulated number of generated strings can get to  $k$ . But if  $|L(r)| < k$ , the total generated strings are equal to  $|L(r)|$ .

For the semantic comparison between  $r_1$  and  $r_2$ , the edit is directional, from  $r_1$  to  $r_2$ . To give the total set of all strings in both languages, we compute the union set  $L = L(r_1) \cup L(r_2)$ . This removed duplicates so only unique strings are considered for the analysis (i.e., if the string  $s \in L(r_1)$  and  $s \in L(r_2)$  for the same string  $s$ , we want to count this once). Then, we matched each string in  $L$  with  $r_1$  and  $r_2$  separately, to form sets  $A$ ,  $B$ , and  $C$ , as described previously.

#### 5.4.4.1 Limitations

For the semantic analysis, we note that a regular expression  $r$  could be invalid in Java regular expression syntax, or Rex may not be able to generate strings for  $r$  due to feature limitations. If either  $r_1$  or  $r_2$  is invalid in Java or contains features beyond the scope of Rex capabilities, we skipped semantic analysis. For example, an edit chain of length five  $r_1, r_2, r_3, r_4, r_5$  should have four semantic comparisons. If  $r_3$  is invalid in Java syntax, this edit chain reduces to two comparisons, between  $r_1$  and  $r_2$ , and between  $r_4$  and  $r_5$ .

For infinite languages, and languages larger than the upper bound on strings generated (i.e.,  $k = 500$ ), the approach described in Section 5.4.2 describes an optimistic approximation of the actual similarity between regular expressions. This is a relatively common scenario due to the common presence of the KLEENE star and ADD operator in a majority of the regular expressions (see Table 5.4). The only sound classification is overlap; all other classifications might be the result from ignoring one or two words in one (or both) of the languages. We depend on Rex for the string generation to determine similarity, and thus inherit any of its biases.

---

<sup>5</sup><https://commons.apache.org/proper/commons-text>

## 5.5 Artifacts

We address research questions in this chapter using two datasets. One comes from GitHub commit logs, providing a high-level view of regular expression evolution over time. The other comes from screencasts of students solving regular expression tasks in an IDE, providing a low-level view of evolution during problem-solving tasks.

### 5.5.1 GitHub Dataset

The history of regular expressions in GitHub projects is collected through source code commits. Since only literal regular expressions can be found statically, we are limited to the explicitly written regular expressions.

#### 5.5.1.1 Data Collection

Our data collection starts within the 1,114 Java projects used in a prior work on testing regular expression [199]. We first searched for method invocations of `Pattern.compile(String regex)` in latest source code version. If the argument is a literal string (as opposed to a variable), we follow the commit history of the literal string to create an edit chain for the regular expression. We filtered out the invocations in which the argument of `Pattern.compile` contains variables and extracted the files and the line numbers where literal regular expressions appear. For other methods, `String.matches(String regex)` needs to check if the caller is a String instance, and `Pattern.matches(String regex, CharSequence input)` contains two arguments that can vary independently; for this first data-driven exploration of regular expression evolution, we focus on the `Pattern.compile()` method.

There are 9,952 static invocations to `Pattern.compile()` in the 1,114 projects; 387 (34.74%) projects contain no literal regular expressions in their latest version and were excluded, resulting in 4,156 literal regular expressions in 727 GitHub projects; these are the tops of the edit chains.

#### 5.5.1.2 Building the Edit Chains

Next, in order to retrieve the commit history of each literal regular expression, we used the Git command `git log -L <start>,<end>:<file>`. We retained information regarding the regular expression edit, commit number, author, and date of each regular expression version. We dropped 194 (4.67%) chains for the following reasons: 1) 123 were dropped

because more than one regular expressions are changed in a single commit on the chain; 2) 30 were dropped because non-literal regular expressions exist in their history of commit changes; 3) 24 were dropped because at least one of the invocations to `Pattern.compile()` are multi-line statements and we failed to parse them; and 4) 17 were dropped because their `git log -L` commands return git activities (e.g., merging files) rather than code edits.

If two adjacent regular expressions are identical to each other in syntax (e.g., something else on the source code line changed, such as a variable name), then this regular expression does not evolve; we squashed these into a single node on the edit chain. There are 144 pairs of such regular expressions. As a result, in the GitHub dataset for study, there are 3,962 edit chains containing 4,224 regular expressions from 708 GitHub projects.

## 5.5.2 Video Dataset

We ran an exploratory lab study in which participants completed regular expression tasks in Java using the Eclipse IDE. During problem solving, we captured videos of their computer screens. Participants were free to use online resources to help them complete the tasks.

### 5.5.2.1 Tasks

Participants attempted up to 20 tasks each, with one hour allotted for the study. The order of tasks was randomized per participant to control for learning effects. In each task, the goal was to compose a regular expression that caused an associated JUnit test suite to pass. For example, one task asked participants to compose a regular expression that will *verify that an entire string is composed of one valid email. Extra characters like whitespace before or after, or anything that would invalidate the email are not allowed.* For this task, eight test cases are provided to demonstrate the desired matching behavior. One test case provides the test input `"name@domain.com"` with the expected output `true`, as in the e-mail address is valid. Another test case has the test input `"1.2.3.4@crazy.domain.axes"` and output `true`. For invalid examples, `"www.website.com"` has the expected output `false`. A repo with all the task data is made publicly available.<sup>6</sup>

---

<sup>6</sup><https://github.com/softwarekitty/regexCompositionStudy/>

### 5.5.2.2 Participants

There were 29 participants who produced usable data for this analysis (six videos had issues with recording). The participants consist of 25 undergraduate students and four graduate students with on average 4.16 years of programming experience and 3.26 years of Java experience. The survey results found that a majority of participants (76%) considered themselves as having intermediate Java programming knowledge; 20 participants (69%) have little to no experience with regular expressions.

### 5.5.2.3 Data Extraction

The videos were manually transcribed to logs reflecting the evolution of the regular expression strings during composition on each problem. In the transcription process, we created a log for each task the participant attempted in each video. Each video was transcribed by one of the authors to ensure consistency within a log. An edit chain consists of all the regular expressions written (or copy/pasted) while attempting to solve a single task. The regular expression logged are the ones submitted by participants for testing.

### 5.5.2.4 Data Description

In total, there are 92 edit chains containing 751 regular expressions from 25 tasks and 29 participants. Twelve pairs of identical adjacent regular expressions are squashed, resulting in 92 edit chains containing 739 regular expressions.

## 5.6 RQ1: Regular Expression Evolution Characteristics

We describe the characteristics of regular expression evolution through analyzing the edit chains in the datasets.

### 5.6.1 Edit Frequency of Regular Expressions

In GitHub dataset, of the 3,962 edit chains containing 4,224 regular expressions. Among the edit chains, 3,775 (95.28%) have a length of one, indicating those regular expressions are not edited at all. Regarding the remaining 187 edit chains of 449 regular expressions,

137 contain one edit, 35 contain two edits, five contain three edits, and ten contain four edits; in total, this created 262 edits.

In Video dataset, of the 92 edit chains containing 739 regular expressions, there are 16 (17.39%) chains of length one. Regarding the remaining 76 regular expression edit chains, 11 contain one edit, eight contain two edits, seven contain three edits, four contain four edits, 12 contain five edits, and the others contain edits from six to 48. The regular expression created by participants needs on average seven changes before task completion or abandonment; this creates 647 edits.

**Summary:** During problem solving, developers tend to modify their regular expressions quite frequently in order to successively complete a task. However, once the related source code committed to the GitHub repositories, edits are rare; 95% of the literal regular expressions we explored were not edited.

### 5.6.2 Runtime Errors

The grammar errors in a normal program source code can be highlighted in the integrated development environment (IDE) and checked during program compilation. However, the grammar errors of regular expressions in source code can only be found during program runtime.

Of the 4,224 regular expressions in the GitHub Dataset, there are nine (0.21%) that produce runtime errors on nine (0.23%) edit chains. These impact three of the edits. Of the 739 regular expressions in the Video Dataset, there are 65 (8.80%) regular expressions that produce runtime errors in 26 (28.26%) edit chains. These impact 85 of the edits.

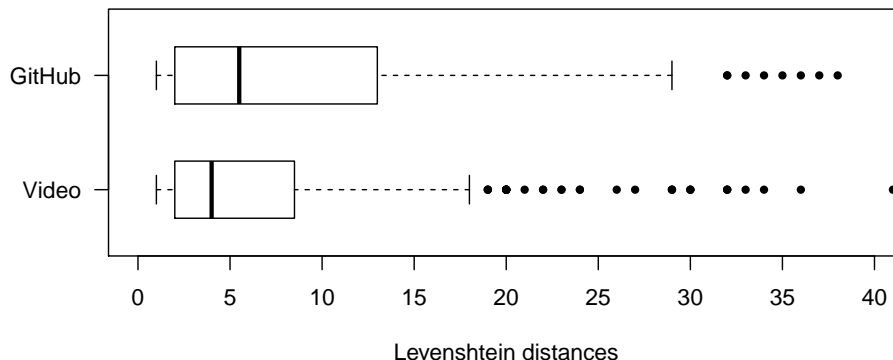
**Summary:** Invalid regular expressions are rarely observed in GitHub; for the video dataset, invalid regular expressions typically usually appear in the early stage of regular expression evolution.

### 5.6.3 Regular Expression Reversions

Regular expression reversion describes the re-occurrence of some regular expressions after they have been modified in an earlier stage. Suppose three regular expressions  $r_i$ ,  $r_j$ , and  $r_k$  ( $i < j < k$ ) on the same edit chain, then the case when  $r_k$  is same as  $r_i$  but different from  $r_j$  is called a regular expression reversion.

**Table 5.1** The distribution of Levenshtein distances in both GitHub and Video edits (where distance > 0).

Dataset	Mean	Min	10%	25%	50%	75%	90%	Max
GitHub	9.32	1	1.00	2.00	5.50	12.75	23.00	52
Video	6.87	1	1.00	2.00	4.00	8.50	15.40	88



**Figure 5.4** The distribution of Levenshtein distances for 262 GitHub edits (with distance > 0) and 647 Video edits (with distance > 0).

In the GitHub dataset, there are 12 regular expression reversions on 12 edit chains; 11 reversions happened in two edits and the other one in three edits.

There are 85 cases of regular expression reversions in Video dataset. Those reversions happened on 25 edit chains; 36 (42.35%) reversions were made in two edits and 11 reversions in three edits. The number of edits in the other 38 reversions varies from four to 28.

**Summary:** Regular expression reversions imply that developers may repeat the same regular expression even if they have previously modified it. This is especially true for inexperienced developers since reversions are more common in the Video dataset; the high frequency possibly reflects developers’ *undo* behavior.

## 5.7 RQ2: Syntactic and Semantic Evolution

We explore RQ2 regarding syntactic and semantic evolution.

### 5.7.1 Syntactic Similarity

We report on Levenshtein distance for the GitHub and Video datasets considering individual *edits*. Starting with an example, one regular expression extracted from GitHub

was committed by user *ginere* in one version<sup>7</sup> and was changed by the same person in a newer version.<sup>8</sup> The original and modified regular expressions are, respectively,

```
\\|DATE\\|([a-zA-Z0-9_]*)\\|\\|
\\|DATE\\|([a-zA-Z0-9:\\\\- /]*)\\|\\|
```

The only syntactic edit is the change from `_` to `:\\- /`, resulting in a Levenshtein distance of five (“`\\`” is considered as one character because backslashes are escaped in Java).

In the study of GitHub dataset on Levenshtein distance, we calculated 262 regular expression edits on 187 edit chains. In Video dataset, we calculated 647 regular expression edits among 76 edit chains.

The edit distance for the GitHub edits is generally larger than the edits from the Videos. Figure 5.4 shows the distribution of Levenshtein distances among the edits in both dataset; Table 5.1 details the averages and distributions of the Levenshtein distances for the 264 edits. On average, there is a distance of 9.32 for a GitHub edit with a median of 5.50, and a distance of 6.88 for a Video edit with the median of 4.00.

**Summary:** The average and median Levenshtein distances in GitHub are larger than in the Video dataset. This reflects our intuition that developers try many small edits while composing and debugging a regular expression. Accordingly, the regular expressions which are committed to version control software reflect larger edits from their predecessors.

From the perspective of regular expression changes, both of the datasets have over 50% regular expression edits in which at most six characters change. Those modifications are the ones could be made automatic through regular expression mutation. However, for the other half of the regular expressions, the changes are much larger. This information suggests that when generating regular expression mutants, we should also consider ways larger changes and/or changes in multiple locations.

## 5.7.2 Semantic Similarity

Within each edit chain, we look at each edit and classify it according to the semantic evolution types in Figure 5.1. In order to compute the similarity between regular expressions, we adopt an approach from prior work [31] and use Rex [195].

---

<sup>7</sup><https://github.com/ginere/ginere-site-generator/commit/7c819359>

<sup>8</sup><https://github.com/ginere/ginere-site-generator/commit/248d3a25>

**Table 5.2** Edit types in GitHub and Video dataset when  $k = 500$ .

Dataset		Disjoint	Overlap	Equivalent	Reduction	Expansion	Total
GitHub	count	44	17	22	20	106	209
	(%)	21.05	8.13	10.53	9.57	50.72	100.00
Video	count	125	32	36	43	56	292
	(%)	42.81	10.96	12.33	14.73	19.18	100.00

**Table 5.3** The distribution of Intersection, Addition, Removal percentages over all types of regular expression edits in GitHub and Video dataset when  $k$  is 500.

Dataset		Mean	Min&10%	25%	50%	75%	90%&Max
GitHub	Intersection	56.62	0.00	29.30	70.37	83.05	100.00
	Addition	38.98	0.00	5.06	26.39	57.58	100.00
	Removal	27.71	0.00	0.00	0.00	52.56	100.00
Video	Intersection	35.78	0.00	0.00	3.87	73.09	100.00
	Addition	56.42	0.00	0.00	79.75	100.00	100.00
	Removal	54.73	0.00	0.00	53.49	100.00	100.00

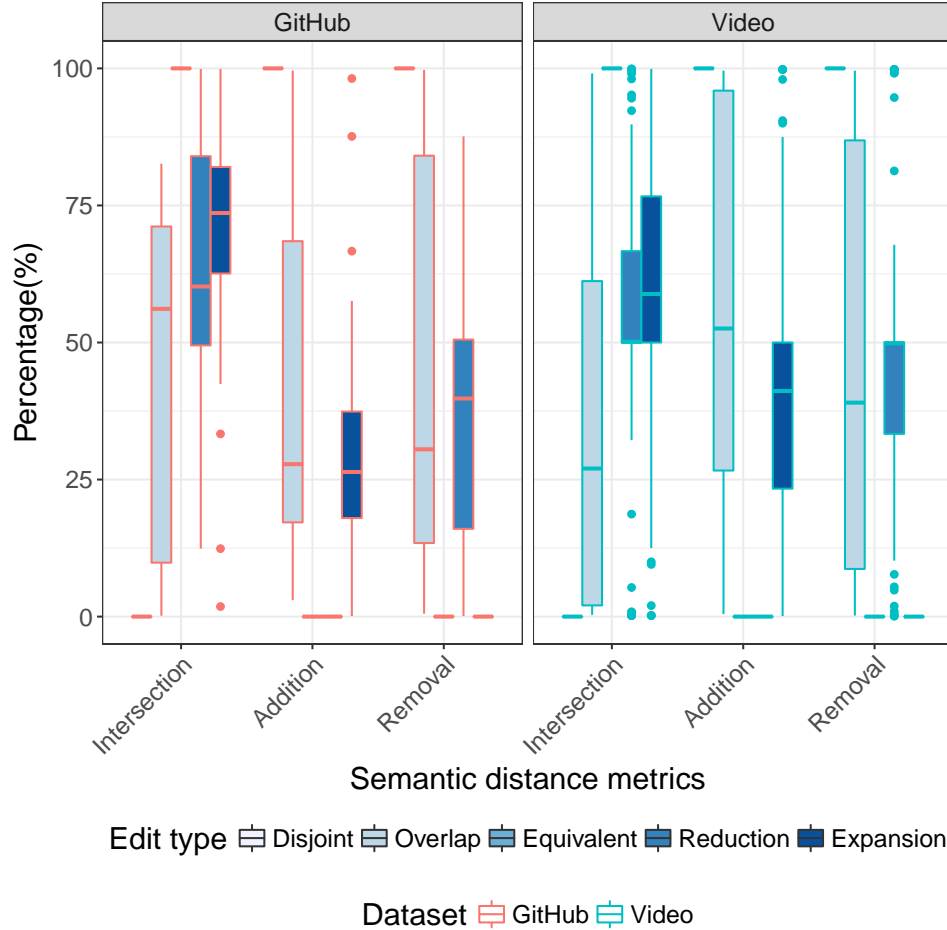
For strings generated by Rex, we chose to generate up to  $k = 500$  strings for each regular expression, as described in Section 5.4.4. This is 25% larger than prior work, which generated 400 strings to compute clusters of similar regular expressions [31].

### 5.7.2.1 Rex-generated Strings

With  $k = 500$ , Rex tries five times with different seeds to generate 500 matching strings for every valid regular expression. The number of matching strings for regular expression  $r_i$  can be different from that for  $r_{i+1}$  because Rex could not guarantee to generate exactly 500. For the semantic distance between  $r_i$  and  $r_{i+1}$ , it requires that both are valid regular expressions and Rex are able to generate matching strings for both of them. Due to the invalid regular expressions next to the valid ones and chains of only one valid regular expressions, the total number of regular expressions involved in the semantic distance is fewer than the ones for which Rex can generate strings.

For the GitHub dataset, Rex generated matching strings for 441 out of the 446 valid regular expressions; 272 have 500 rex-generated matching strings while the other 169 have fewer than 500 matching strings. On average there are 432 matching strings generated for each regular expression. In total, 209 edits on 146 edit chains are studied for the GitHub dataset.

In the Video dataset, Rex generated matching strings for 393 out of 674 valid regular



**Figure 5.5** The distribution of Add, Remove, and Overlap percentages over disjoint, equal, overlap, subset and superset regular expression edits in GitHub and Video dataset when k is 500.

expressions; 102 regular expressions have 500 strings and the other 291 ones have fewer than 500 strings. On average there are 270 Rex-generated matching strings per regular expression. In total, 292 edits on 47 edit chains are studied for the Video dataset.

### 5.7.2.2 Results

Table 5.2 shows the number of different edit types in the GitHub and Video datasets. The most common edit is *expansion* for GitHub (50.72%) and *disjoint* for Video (42.81%). Table 5.3 shows the distribution of intersection, addition, and removal metrics for every regular expression edit in both datasets. The average intersection value for an edit in the GitHub dataset is 56.62%, indicating that more than half of the language from an edited regular expression is sourced from its predecessor. This makes sense as a majority of the edits fall in the *expansion* classification (Table 5.2). In the Video dataset, only

35.78% of the language of a regular expression is sourced from its predecessor, likely lower because of the high frequency of *disjoint* edits. In the GitHub dataset, the relatively small addition and removal values indicate that the semantic change is relatively small per edit, whereas with the addition and removal values in the Video dataset are larger, on average, indicating larger semantic changes per edit.

The GitHub dataset edits represent the situations where the regular expression being modified are close to the targeted scope of strings because of their small number of edits and a high percentage of edit intersection. This indicates that the semantic edits are relatively small.

The differences can also be observed in Figure 5.5 which visualizes the distributions of metrics per semantic edit type. In all edit types, the intersection value of GitHub dataset is higher than that of Video dataset while the addition and removal values of GitHub dataset are always lower than of Video dataset. In the *expansion* edit type for GitHub, the average addition is 29.02%; for the video analysis, the addition is 41.52% on average. *Reduction* edits in GitHub remove an average of approximately one-third (35.55%) of the language, whereas the Video reductions remove an average of 47.08%. Average intersection numbers for *overlap*, *reduction*, and *expansion* in the GitHub edits are 45.08%, 64.45%, and 70.98% whereas the average intersection numbers for these three edit types in the Video edits are 36.67%, 52.92%, and 58.48%.

## Refactoring Analysis

We did a further study on pairs of regular expressions classified as *equivalent*. In the GitHub dataset, 19 out of the 22 pairs are correctly classified. For the three misclassified cases, one changes the repetition time from `[a-z0-9_-]{1,64}` to `[a-z0-9_-]{1,120}`. This misclassification is because the length of strings generated by Rex is less than 13. The other two cases change special characters in the character class, and Rex does not use those special characters in the string generation.

Among the 19 truly equivalent pairs, eight pairs are related to unnecessary character escaping (e.g., from `([\\+\\-])+(.*)` to `([+\\-])+(.*)`) and three pairs are related to changes in capturing group representation (e.g., from `.* \\{.*\\}` to `(.*) (\\{.*\\})`). One removes capital characters from `[aA][sS]` and expresses the case sensitiveness with flag `(?i)AS`, and two adds and removes capital characters in `(?i)[0-9a-fA-F]` and `(?i)[0-9a-f]` while regular flag ‘i’ is specified for case-insensitive matching. One changes character repetition boundary `[0-9]{1,}` to greedy operator `[0-9]+`, hence

improving improves the understandability according to a regular expression comprehension study [32]. One changes the literal parentheses in character class from `[()]` to escaped characters `\\(\\)`. One changes the ordering on options surrounding an OR operator, from `([wdhms]|ms)` to `(ms|[wdhms])`. For full matches (as is the case with `Pattern.matches()`), these are identical. The final two pairs change whitespace around a `.*`, from `(?im)^dry-run:\\s*(.*)\\s*` to `(?im)^dry-run:(.*)` and from `(?im)^dry-run:(.*)` to `(?im)^dry-run:(.*)\\n*`. Effectively, these are all equivalent.

In the Video dataset, 35 of the 36 pairs are correctly classified. The only misclassified case changes the whitespace characters from `00Z*([a-zA-Z\\s]*)` to `00Z*([a-zA-Z ]*)`. It is correctly classified as *reduction* when  $k = 1000$ .

Among the 35 truly equivalent pairs, 12 pairs are related to changes in capturing group representation, nine pairs are related to unnecessary character escaping, three are about adding or deleting anchors (e.g., from `^[.]+@[a-zA-Z0-9-]+\\. [a-zA-Z0-9-]+$` to `[.]+@[a-zA-Z0-9-]+\\. [a-zA-Z0-9-]+$,`<sup>9</sup> four about changing OR operator alternatives which does not impact matching behaviors, such as removing one option from `(.*)|(6.35.)` to `(.*)`. One removes character class of single element from `[A-Za-z0-9][\\s]` to `[A-Za-z0-9]\\s`. Two pairs manipulate duplicated characters in the character class between `[(1|3|5|7|9)+(2|4|6|8|0)]` and `[(1|3|5|7|9)+(2|4|6|8|0)]` possibly due to misconceptions of the differences between `[]` and `()`. Another edit changes from `.*[02468][13579].*` to `.*([02468][13579]).*`. Although `+` is added, additional digits are accepted by `.*` at the beginning of these two regular expressions. One changes from `[a-zA-Z-'](.*)total[0-9]` to `[a-zA-Z-']+(.*)total[0-9]`. Although `+` is added, additional characters are accepted by `(.*)` in the middle of these two regular expressions. In the modification from `^[a-zA-Z0-9 \\t]*$` to `^[a-zA-Z0-9 \\d \\t]*$`, the regular language does not change because `\\d` is equivalent to `0-9` which exists already in the character class. The final pair changes from `((1|3|5|7|9)+(2|4|6|8|0))*|((2|4|6|8|0)+(1|3|5|7|9))*` to `((1|3|5|7|9)(2|4|6|8|0))+|((2|4|6|8|0)(1|3|5|7|9))+`. This is because `a|b` contains three alternatives: `a`, `b`, and empty strings and `(1|3|5|7|9)` matches not only odd digits but also empty strings. The changes of repetitions in these two regular expressions are thus counteracted by the empty strings.

---

<sup>9</sup>Since `Pattern.matches(String regex, CharSequence input)`, `String.matches(String regex)`, and `Matcher.matches()` by default match the entire input string to the regular expression and anchors do not affect the matching results.

**Summary:** Compared to the GitHub edits, the edits in the Video dataset tend to make larger semantic changes. GitHub edits represent small adjustments of the regular expression close to targeted scope.

## 5.8 RQ3: Regular Expression Feature Changes

In this section, we present our results and analysis of regular expression language feature changes in the edit chains. This can inform the features to focus on during mutation testing or program repair.

The feature vector we use contains the 35 most frequently used features in Java regular expressions. All feature explanations except *LIT* (literal character) are defined in the work of Chapman and Stolee [31]. Features are extracted using the PCRE parser.

For the GitHub dataset, we started with 262 edits; 3 were removed due to Java runtime errors (Section 5.6.2) and three were removed due to PCRE parsing errors, leaving us with 256 edits for analysis. For the Video dataset, we started with 647 edits; 85 were removed due to runtime errors and four were removed due to PCRE parse errors, resulting in 558 edits for analysis.

### 5.8.1 Feature Vector Edits

We first calculated the number of regular expressions in which each feature appears. For the 4,054 regular expressions in GitHub and 660 regular expressions in Video Feature that can be parsed by PCRE. Table 5.4 shows the results of frequency analysis for the top 25 features in the *Freq* column for the GitHub and Video Datasets.

For 237 of the GitHub edits and 536 of the Video edits, the feature vector (e.g., Figure 5.3) changed.  $F_{add}$  and  $F_{remove}$  are listed, alongside the number of edits impacted ( $nR$ ). For example, 123 edits in GitHub added a literal (LIT), and 53 added a KLEENE star (KLE). The ADD feature is the third most commonly seen feature in the GitHub dataset. It is added into 34 regular expressions and ranked as the sixth most frequently added feature while it is removed from 27 regular expressions and ranked as the second most frequently removed features. Assuming that the predecessor in a regular expression edit has a fault of some sort, these details can help inform the types of changes to make to a regex feature vector during fault injection for mutation testing.

Overall for every feature in both datasets, there are more regular expressions which added it than the ones which removed it. From Table 5.5 we can find that feature frequency

**Table 5.4** Statistics of the language features in GitHub and Video dataset ranked by the number of regular expressions in which features present (Freq), features are added, and features are removed.

rank	GitHub					Video				
	Freq	Add	nR	Remove	nR	Freq	Add	nR	Remove	nR
1	LIT	LIT	123	LIT	62	LIT	LIT	191	LIT	144
2	CG	KLE	53	ADD	27	CG	CG	95	CG	63
3	ADD	QST	42	KLE	24	ANY	KLE	76	KLE	53
4	KLE	ANY	36	QST	23	KLE	ANY	55	ANY	40
5	CCC	CG	34	CCC	22	ADD	ADD	54	ADD	37
6	ANY	ADD	34	CG	21	CCC	CCC	33	CCC	24
7	RNG	CCC	33	ANY	17	WSP	WSP	27	WSP	23
8	STR	RNG	24	DEC	12	RNG	OR	24	OR	15
9	END	OR	20	RNG	11	OR	STR	23	STR	15
10	DEC	NCG	13	END	8	WRD	DEC	21	WRD	12
11	QST	NWSP	9	STR	8	DEC	LZY	16	RNG	11
12	NCCC	NCCC	9	NWSP	7	END	QST	15	DEC	10
13	WSP	WSP	8	OR	6	STR	END	13	END	10
14	OR	WRD	8	NCG	5	LZY	RNG	12	NCCC	10
15	WRD	DEC	7	LWB	5	QST	WRD	12	LZY	8
16	LZY	END	6	DBB	4	WNW	WNW	11	WNW	8
17	SNG	STR	5	LZY	3	NCCC	NCCC	10	QST	7
18	NCG	LZY	4	NCCC	2	SNG	SNG	10	SNG	3
19	NWSP	DBB	3	WSP	2	LKB	LKB	4	LKB	3
20	DBB	OPT	3	WRD	2	BKR	NWRD	3	NCG	3
21	OPT	NDEC	2	SNG	2	NCG	NDEC	2	NLKA	2
22	LWB	LKA	2	LKA	1	LWB	NCG	2	LKA	2
23	WNW	LWB	1	OPT	0	NLKA	BKR	2	LWB	1
24	LKA	SNG	1	NDEC	0	NDEC	LWB	2	NWNW	1
25	NWRD	NWRD	0	NWRD	0	LKA	NLKA	2	NDEC	0

is different between GitHub and Video. For example, on the average edit, 3.71 features are added to a regular expression and 2.52 are removed. For the video dataset, these values are smaller, with 2.60 features added and 1.98 removed. These details can help inform the frequencies of changes to make to a feature vector during fault injection for mutation testing or repair.

### 5.8.2 Feature Vector Non-Edits

The feature vector used in this study does not reflect the positions of features, nor the scope of the changes. For example, the modification from `[\\D]{2}` to `[\\D]{5}` does not change the number of feature SNG but it changes the repetition time of SNG from ‘2’

**Table 5.5** Distribution of regular expression feature changes among 256 edits in GitHub dataset and 558 edits in Video dataset.

Dataset		Mean	Min	10%	25%	50%	75%	90%	Max
GitHub	$F_{add}$	3.71	0	0	1	2	5	9	37
	$F_{remove}$	2.52	0	0	0	0	2	7	37
	$F_{add}+F_{remove}$	6.23	0	1	2	4	8	15	39
Video	$F_{add}$	2.60	0	0	0	1	3	6	73
	$F_{remove}$	1.98	0	0	0	1	2	5	66
	$F_{add}+F_{remove}$	4.58	0	1	1	2	5	11	75

to ‘5’. Similarly, the change from  $(^{\backslash}r\backslash f\backslash n)$  to  $^{\backslash}(r\backslash f\backslash n)$  changes the content of capturing group to exclude ‘^’, but the feature vector remains the same.

There were 19 edits in the GitHub dataset and 22 in the Video dataset for which the vectors did not change. On further inspection, the most common modification is related to backslash for character escaping (e.g., from  $\backslash t\backslash n$  to  $\backslash\backslash t\backslash n$ ), impacting 10 edits in the GitHub dataset and 13 edits in the Video dataset. Other common modifications change characters to other characters (e.g., from  $\backslash\{([\backslash w\backslash.]*)\backslash\}$  to  $\backslash\{([\backslash w\backslash.]*)\backslash\}$ ), switch the order of characters (e.g., from  $\backslash f\backslash\backslash s$  to  $\backslash\backslash s\backslash f$ ), change repetition times (e.g., from  $[a-z0-9_-]\{1,64\}$  to  $[a-z0-9_-]\{1,120\}$ ), change the content and position of capturing group (e.g., from  $(^{\backslash}r\backslash f\backslash n)$  to  $^{\backslash}(r\backslash f\backslash n)$ ), change characters in the character class (i.e., from  $^{\backslash}[a-zA-Z0-9\backslash\backslash\backslash]*\$$  to  $^{\backslash}[a-zA-Z0-9\backslash\backslash t]*\$$  or change alternation option (e.g., from  $^{\backslash}(r\backslash\backslash n)$  to  $^{\backslash}(r\backslash\backslash f)$ ).

**Summary:** The regular expression edits usually involve changing multiple features; adding features is more common than removing. The frequency of various features being added and being removed can be used to construct new regular expression mutation operator and guide mutation generation process. However, there are also edits that do not impact the feature vectors; escaping characters is the most common edits do not result in changes in the feature vector.

## 5.9 Discussion

In this chapter, we have looked at the evolution of regular expressions from two perspectives, syntactic and semantic, and in two contexts, using GitHub and developers solving tasks.

### 5.9.1 Implications

Most literal regular expressions in GitHub do not evolve (Section 5.6), and yet, bug reports related to regular expressions abound [179]. This may indicate that buggy regular expressions are simply removed from source code, or another solution to avoid the use of a regular expression.

Yet, regular expressions are under-tested [199], and most string-generation efforts focus on generating test inputs within the language of the regular expression (e.g., [195]). For those regular expressions that do evolve, 50% of the edits in GitHub are expansion edits (Section 4.6.1), indicating that often the original regular expression language is too restrictive. In generating test inputs, there is a need for test strings that lie outside the language of the original regular expression. One approach to this is fault injection via mutation [9]. However, to do this effectively, these faults need to be reflective of edits that developers make to regular expressions.

For those regular expressions that do evolve, 50% of the edits have a syntactic distance of six or fewer characters; these are most amenable to mutation testing (Section 4.6.1). Edits also impact multiple language features (Table 5.5).

For the Video dataset, the edits tend to be smaller in terms of character modifications (Table 5.1), but larger in terms of semantic distancing (Table 5.3 shows the intersection for Video edits is smaller than for GitHub edits). Furthermore, the edits to the feature vector tend to be smaller for the Video dataset (Table 5.5). The Video dataset edit chains are also longer than the GitHub edit chains (Section 5.5). Even though the Video dataset participants were largely novices, this indicates that developers likely go through many smaller iterations of edits on the regular expressions before finding one to commit.

### 5.9.2 Threats to Validity

**Internal:** We measure similarity using a string-generation approach that provides an approximate measure of similarity. We also observed in Section 5.7.2 that while Rex is a well-used and well-cited tool, sometimes it did not generate strings long enough and strings for uncommon characters. Such tool limitations have an impact on the accuracy of our results.

The regular expressions from the video analysis were collected manually. Each video was transcribed by two graduate students and merged to address any inconsistencies.

We collect regular expressions from GitHub using the current version of a project at

the top of the chain. Regular expressions that are removed previously in the edit history will not be included in our dataset.

**External:** The regular expressions collected in this project reflect a relatively small sample, in one language (Java), and may not generalize. Further, the Video dataset was collected in a lab environment and may not reflect how developers actually compose regular expressions. While further study is needed, we do note that the common language features are consistent with prior work that explored regular expressions in another language [31].

The literal regular expressions we collected are restricted to `Pattern.compile`. Regular expressions with explicit flags to that method are excluded as well. Other Java methods which also accept literal regular expressions are not included in GitHub dataset.

## 5.10 Summary

In this chapter, we explore how regular expressions evolve through two lenses, GitHub commits and tested regular expressions during problem-solving tasks (called the Video dataset). We find that the GitHub regular expression edits are larger syntactically, but produce smaller semantic changes. The edit chains in the Video dataset are longer than the GitHub edit chains, indicating that developers may go through multiple iterations prior to committing a regular expression to a repository. The most common change to the scope of the regular expression in GitHub is to expand the matching language, which motivates the use of mutation operators to generate strings outside the original language for testing. Our results provide insights on the types and frequencies of edits that occur in regular expressions and can be used to guide mutation operators that reflect developer practices.

## CHAPTER

# 6

## CONCLUSION

In Chapter 6 we revisit the thesis statement, conclude the discussions in this dissertation, and present some thoughts on future work according to these discussions.

### 6.1 Thesis Statement Revisited

This dissertation presents research to evaluate and support my thesis statement (Chapter 1). The thesis of this dissertation is:

Regular expressions in open source software have **low test coverage**, are modified **infrequently**, are overly **restrictive**, and contribute to software **bugs**.

To support the claim that *regular expression contribute to software bugs*, I conducted a *qualitative* study of 356 merged regular expression related pull requests from 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 (Chapter 3). 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. This study demonstrates that regular expression bugs are not independent of the source code but are influenced by the software evolution and the code quality. Those 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. The analysis on test code changes indicates that regexes may be under-tested.

To defend the claim that *regexes have low test coverage*, I conducted another *quantitative* analysis of regular expression testing coverage in GitHub Java repositories and evaluated how effective string generation tools are in testing regexes (Chapter 4). The results show that over 80% of *FullMatch* functions are not tested and that most of the tested regular expressions have a low edge and edge-pair coverage. The results also show that with the help of the regular expression tool Rex, it is possible to improve the regular expression testing coverage by adding input strings, but that there is an upper bound for this type of improvement.

To demonstrate that *regexes are overly restricted and infrequently modified*, I examined the regular expression edits through two lenses, GitHub commits and tested regular expressions during problem-solving tasks (Chapter 5). It is found that the 95% of the regular expressions from GitHub are not edited. For the edited regexes, over 50% of the edits in GitHub tend to expand the scope of regular expression, and the number of features used indicates the regular expression language usage increases over time.

In this dissertation, I explore regular expression usage in different software development processes: testing [199, 201], debugging and bug fixing [200], refactoring [200], and software evolution [198]. I analyze the real-world problems developers have to deal with using both quantitative and qualitative studies.

## 6.2 Future Work

The future work in this dissertation is motivated by different studies. The study of regex testing coverage (Chapter 4) motivates the future work on exploring how various coverage metrics can reduce the bugs associated with regular expressions. The findings in the regex bug study (Chapter 3) provide an overview of regular expression bugs and motivates future work on techniques and tools to solve practical regular expression bugs. In the regex evolution study (Chapter 5), the most common change to the scope of the regular expression in GitHub is to expand the matching language, which motivates the use of mutation operators to generate strings outside the original language for testing.

Our results provide insights on the types and frequencies of edits that occur in regular expressions and can be used to guide mutation operators that reflect developer practices. Finally, given the observations and findings of regex usage in software development, we shift the focus from software development to the developers and share some thoughts for the developers' perspective.

### 6.2.1 Beyond Structural Coverage Metrics

The metrics explored in Chapter 4 are structural metrics, which can identify faults that are revealed in the structure of the DFA, such as the example in Section 4.2.4. Alternately, as suggested in prior work [32], refactoring could potentially reveal this particular fault, as the numeric representation `[0-9]` was found to be more understandable than `\d`. Performing the replacement might alert the developer that `d` should be `\d`.

In terms of improving regular expression testing, structural metrics are a first step. Building on the example in Section 4.3, achieving 100% coverage requires a minimum number of test inputs that vary in string length and content. In the example of `\d+`, there are strings of length one to length four, though strings could be longer to test multiple iterations on the self-loop. Strings can contain only digits, only non-digits, or both digits and non-digits. Strings can start with digits or start with non-digits. Defining such input space partitions may lead to intuitive test sets with high behavioral coverage.

### 6.2.2 String-generation Tools

Given the low coverage of regular expressions shown in Figure 4.7, a natural next step could be to generate strings to achieve high coverage. Adding a mutation step to the input string may be effective at forcing the Rex-generated strings into the error state to cover the uncovered edges and node. An alternate approach may be to provide the complement of the regular expression to Rex as another way to generate failing inputs.

The intersection of regular expression edits is less than 60%, indicating the string-generation tool should also consider generating a certain percentage of strings not matching to the regular expression but matching to its mutants.

With automatically-generated strings, one threat is usability. For the developer-written tests, it is likely that the regular expression strings are more meaningful in context than they are for the Rex generated strings. Future work will look at the overlap in content between the test inputs from the repository and from Rex. However, it may not always

be possible to achieve 100% test coverage, even with a perfect string generation tool. Some regular expressions have hard-coded matching inputs, which makes it impossible to improve the coverage; for example: `boolean isMatch = Pattern.matches("a*b", "ab");`. Future work for improving coverage levels should also consider the potential for improvement based on such factors.

## 6.2.3 Robust Regular Expression

Coverage provides useful stopping criteria for testing. However, high coverage does not necessarily imply test suite effectiveness in source code [83], which may also hold true for regular expressions. Coverage neither provides any helpfulness to evaluating whether the regex usage is optimized. At the same time, as regular expressions are responsible for many software issues, it is important to explore how to make them less error-prone and more efficient.

### 6.2.3.1 Mutation Testing

One big motivation of studying regular expression evolution is to apply empirical knowledge to mutation testing of regular expressions. The difference between GitHub and Video dataset suggest we try different mutation strategies according to the stage of software development. The mutation operators can be defined depending on the changes among regular expression features and inside each regular expression feature. The priority of various mutants can be ranked according to their syntactic and semantic distance.

### 6.2.3.2 Rule-based Bug Detection, Regex Refactoring, Repairing

During the analysis of regular expression changes or API changes, some regular expression fix patterns are found to be recurring. It is the same with the regex optimization patterns. Applying those patterns to identify similar code prior to the change at a large scale could unveil the prevalence of the types of regular expressions bugs we have discussed in this paper. More importantly, applying those patterns to fix those bugs and integrating such tools into software development can ease the burden of developers and help improve the code quality as well.

## 6.2.4 Regular Expressions and Developers

While the research on regular expression could enhance the knowledge on regex usage and improve its correctness, performance, and comprehension, it is also important to train regex users the correct way of using regexes and to adapt the current development environment. Besides teaching the syntax of regular expressions, it is essential to present the correct regex practice on when to use regexes, how to write a correct regex, how to validate it, how to test and document its usage. Regular expression is a context-free language and it could be improper to use regex in the parsing of structural data (e.g., HTML, URL). Telling them when regular expressions should be and should not be used could help prevent the complicated regexes which are hard to understand. Also, knowing the benefits and drawbacks of regex solution could help regex users make better decisions when comparing the regex solution and the alternative solution for string processing issues.

One major drawback of regex tools is that they are usually standalone and independent of the software development process. Merging them into the software development is another way to ease the requirement of using regexes for developers and to prompt developers to use regex tools more frequently. The tools can be added into the integrated development environment (IDE) as plugins so that developers can get immediate feedback about the regexes being used in the source code. Another promising direction for regex optimization and security tools is to make them default in the regular expression engines. Instead of relying on developers to optimize regex and check whether the regex has security vulnerabilities, the regex engine can apply automatically the optimization and remind developers of the potential existence of security risks.

## 6.3 Epilogue

*Some people, when confronted with a problem, think “I know, I’ll use regular expressions.”. Now they have two problems*<sup>1</sup>.

---

<sup>1</sup><http://regex.info/blog/2006-09-15/247>

## BIBLIOGRAPHY

- [1] Abbes, M. et al. “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension”. *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE. 2011, pp. 181–190.
- [2] Amann, S. et al. “MUBench: A benchmark for API-misuse detectors”. *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016, pp. 464–467.
- [3] Amann, S. et al. “A systematic evaluation of static api-misuse detectors”. *IEEE Transactions on Software Engineering* **45.12** (2018), pp. 1170–1188.
- [4] Ammann, P. & Offutt, J. *Introduction to software testing*. Cambridge University Press, 2016.
- [5] *An Empirical Study on Regular Expression Bugs Dataset*. <https://figshare.com/s/802eb74c2e722ca5d8df>. 2020.
- [6] Anand, S., Păsăreanu, C. & Visser, W. “JPF–SE: A symbolic execution extension to java pathfinder”. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), pp. 134–138.
- [7] Anand, S. et al. “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation”. *J. Syst. Softw.* **86.8** (2013), pp. 1978–2001.
- [8] Anand, S. et al. “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation”. *J. Syst. Softw.* **86.8** (2013), pp. 1978–2001.
- [9] Arcaini, P., Gargantini, A. & Riccobene, E. “Mutrex: A mutation-based generator of fault detecting strings for regular expressions”. *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 87–96.
- [10] Arcaini, P., Gargantini, A. & Riccobene, E. “Interactive Testing and Repairing of Regular Expressions”. *IFIP International Conference on Testing Software and Systems*. Springer, 2018, pp. 1–16.
- [11] Arcaini, P., Gargantini, A. & Riccobene, E. “Fault-based test generation for regular expressions by mutation”. *Software Testing, Verification and Reliability* (), e1664.
- [12] Arslan, A. N. “Multiple sequence alignment containing a sequence of regular expressions”. *Computational Intelligence in Bioinformatics and Computational*

*Biology, 2005. CIBCB'05. Proceedings of the 2005 IEEE Symposium on.* New York, NY, USA: IEEE, 2005, pp. 1–7.

- [13] Babbar, R. & Singh, N. “Clustering Based Approach to Learning Regular Expressions over Large Alphabet for Noisy Unstructured Text”. *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data. AND '10.* Toronto, ON, Canada: ACM, 2010, pp. 43–50.
- [14] Babbar, R. & Singh, N. “Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text”. *Proceedings of the fourth workshop on Analytics for noisy unstructured text data.* ACM, 2010, pp. 43–50.
- [15] Backurs, A. & Indyk, P. “Which Regular Expression Patterns Are Hard to Match?” *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS).* 2016, pp. 457–466.
- [16] Bae, S. et al. “SAFEWAPI: Web API misuse detector for web applications”. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 2014, pp. 507–517.
- [17] Baeza-Yates, R. A. & Gonnet, G. H. “Fast Text Searching for Regular Expressions or Automaton Searching on Tries”. *J. ACM* **43.6** (1996), pp. 915–936.
- [18] Bartoli, A. et al. “Automatic generation of regular expressions from examples with genetic programming”. *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation.* ACM, 2012, pp. 1477–1478.
- [19] Bartoli, A. et al. “Inference of regular expressions for text extraction from examples”. *IEEE Transactions on Knowledge and Data Engineering* **28.5** (2016), pp. 1217–1230.
- [20] Becchi, M. & Cadambi, S. “Memory-efficient regular expression search using state merging”. *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications.* New York, NY, USA: IEEE, 2007, pp. 1064–1072.
- [21] Becchi, M. & Crowley, P. “Efficient Regular Expression Evaluation: Theory to Practice”. *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems.* ANCS '08. San Jose, California: ACM, 2008, pp. 50–59.
- [22] Beck, F. et al. “Regviz: Visual debugging of regular expressions”. *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 504–507.

- [23] Beller, M., Gousios, G. & Zaidman, A. “Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub”. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 356–367.
- [24] Brodie, B. C., Taylor, D. E. & Cytron, R. K. “A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching”. *33rd International Symposium on Computer Architecture (ISCA’06)*. 2006, pp. 191–202.
- [25] Brown, W. J., Malveau, R. C. & Brown, W. H. “McCormick,; Hays W., III; Mowbray, Thomas J”. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1998).
- [26] Bucur, S. et al. “Parallel symbolic execution for automated real-world software testing”. *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 183–198.
- [27] Cadar, C., Dunbar, D., Engler, D. R., et al. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” *OSDI*. Vol. 8. 2008, pp. 209–224.
- [28] Callaú, O. et al. “How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk”. *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 23–32.
- [29] Callaú, O. et al. “How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk”. *Empirical Software Engineering* **18.6** (2013), pp. 1156–1194.
- [30] Cetinkaya, A. “Regular Expression Generation Through Grammatical Evolution”. *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO ’07. London, United Kingdom: ACM, 2007, pp. 2643–2646.
- [31] Chapman, C. & Stolee, K. T. “Exploring regular expression usage and context in Python”. *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 282–293.
- [32] Chapman, C., Wang, P. & Stolee, K. T. “Exploring regular expression comprehension”. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 405–416.
- [33] Chapman, C., Wang, P. & Stolee, K. T. “Exploring Regular Expression Comprehension”. *Proceedings of the 32Nd IEEE/ACM International Conference*

on *Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 405–416.

- [34] Chen, T. “Indexing Text Documents for Fast Evaluation of Regular Expressions”. PhD thesis. The University of Wisconsin-Madison, 2012.
- [35] Chiba, S. “Javassist-a reflection-based programming wizard for Java”. *Proceedings of OOPSLA’98 Workshop on Reflective Programming in C++ and Java*. Vol. 174. 1998.
- [36] Cody-Kenny, B. et al. “A search for improved performance in regular expressions”. *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM. 2017, pp. 1280–1287.
- [37] Coelho, R. et al. “Unveiling exception handling bug hazards in Android based on GitHub and Google code issues”. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 134–145.
- [38] *How to Contribute – React*. <https://reactjs.org/docs/how-to-contribute.html>. 2021.
- [39] *google/guava - How to contribute*. <https://github.com/google/guava/blob/master/CONTRIBUTING.md>. 2020.
- [40] *apache/airflow – Contribution Workflow*. <https://github.com/apache/airflow/blob/main/CONTRIBUTING.rst>. 2021.
- [41] Cox, R. *Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...)* <https://swtch.com/%7Ersc/regexp/regexp1.html>. 2007.
- [42] Cox, R. *Regular expression matching in the wild*. <https://swtch.com/~rsc/regexp/regexp3.html>. 2010.
- [43] Crosby, S. “Denial of service through regular expressions” (2003).
- [44] Crosby, S. A. & Wallach, D. S. “Denial of Service via Algorithmic Complexity Attacks.” *USENIX Security Symposium*. 2003, pp. 29–44.
- [45] Davis, J. C. et al. “Testing regex generalizability and its implications: A large-scale many-language measurement study”. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 427–439.
- [46] Davis, J. C. et al. “Why aren’t regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions”. *Proceedings of the*

- 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM. 2019, pp. 443–454.
- [47] Davis, J. C. et al. “The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale”. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 246–256.
- [48] De Moura, L. & Bjørner, N. “Z3: An efficient SMT solver”. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.
- [49] Di Franco, A., Guo, H. & Rubio-González, C. “A comprehensive study of real-world numerical bug characteristics”. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 509–519.
- [50] Dig, D. & Johnson, R. “How do APIs evolve? A story of refactoring”. *Journal of software maintenance and evolution: Research and Practice* **18.2** (2006), pp. 83–107.
- [51] Du Bois, B. et al. “Does god class decomposition affect comprehensibility?” *IASTED Conf. on Software Engineering*. 2006, pp. 346–355.
- [52] Dulucq, S. & Touzet, H. “Analysis of tree edit distance algorithms”. *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2003, pp. 83–95.
- [53] Dyer, R. et al. “Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, 2014.
- [54] Eghbali, A. & Pradel, M. “No Strings Attached: An Empirical Study of String-related Software Bugs”. *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering*. ASE '20. 2020.
- [55] Facebook. <https://github.com/facebook>. 2020.
- [56] Ficara, D. et al. “An improved DFA for fast regular expression matching”. *ACM SIGCOMM Computer Communication Review* **38.5** (2008), pp. 29–40.
- [57] Fixed the RegEx. <https://github.com/CaszGamerMD/NootSpeak/pull/14/>.

- [58] Fowler, M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [59] Fraser, G. et al. “Does automated white-box test generation really help software testers?” *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 291–301.
- [60] Friedl, J. E. *Mastering regular expressions*. " O'Reilly Media, Inc.", 2002.
- [61] Galler, S. J. & Aichernig, B. K. “Survey on Test Data Generation Tools”. *Int. J. Softw. Tools Technol. Transf.* **16.6** (2014), pp. 727–751.
- [62] Garofalakis, M., Rastogi, R. & Shim, K. “Mining sequential patterns with regular expression constraints”. *IEEE Transactions on knowledge and data engineering* **14.3** (2002), pp. 530–552.
- [63] Ghosh, I. et al. “JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings”. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 992–1001.
- [64] Giger, E., Pinzger, M. & Gall, H. C. “Comparing fine-grained source code changes and code churn for bug prediction”. *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 83–92.
- [65] *Github GraphQL API v4 2019*. <https://developer.github.com/v4/>. 2020.
- [66] *GitHut - Programming Languages and GitHub*. <https://githut.info/>. 2014.
- [67] González-Pardo, A. et al. “A Case Study on Grammatical-Based Representation for Regular Expression Evolution”. *Trends in Practical Applications of Agents and Multiagent Systems*. Ed. by Demazeau, Y. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 379–386.
- [68] *Google*. <https://github.com/google>. 2020.
- [69] Gousios, G., Pinzger, M. & Deursen, A. v. “An exploratory study of the pull-based software development model”. *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 345–355.
- [70] Gousios, G. & Zaidman, A. “A dataset for pull-based development research”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 368–371.
- [71] Grechanik, M. et al. “An Empirical Investigation into a Large-scale Java Open Source Code Repository”. *Proceedings of the 2010 ACM-IEEE International*

*Symposium on Empirical Software Engineering and Measurement*. ESEM '10. New York, NY, USA: ACM, 2010.

- [72] Gruber, H. & Holzer, M. “Finite automata, digraph connectivity, and regular expression size”. *International Colloquium on Automata, Languages, and Programming*. Springer. 2008, pp. 39–50.
- [73] Gu, Z. et al. “An empirical study on api-misuse bugs in open-source C programs”. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE. 2019, pp. 11–20.
- [74] Gyimesi, P. et al. “BugsJS: a benchmark of JavaScript bugs”. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 90–101.
- [75] Henglein, F. & Nielsen, L. “Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation”. *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 385–398.
- [76] Hermans, F., Pinzger, M. & Deursen, A. van. “Detecting Code Smells in Spreadsheet Formulas”. *Proc. of ICSM '12*. 2012, pp. 409–418.
- [77] Hermans, F., Pinzger, M. & Deursen, A. van. “Detecting and refactoring code smells in spreadsheet formulas”. English. *Empirical Software Engineering* (2014), pp. 1–27.
- [78] Herzig, K., Just, S. & Zeller, A. “It’s not a bug, it’s a feature: how misclassification impacts bug prediction”. *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, pp. 392–401.
- [79] Hovsepyan, A. et al. “Software Vulnerability Prediction Using Text Analysis Techniques”. *Proceedings of the 4th International Workshop on Security Measurements and Metrics*. MetriSec '12. Lund, Sweden: ACM, 2012, pp. 7–10.
- [80] *How to use a search regex for tags in the Stack Exchange API*. <https://meta.stackexchange.com/questions/332224/how-to-use-a-search-regex-for-tags-in-the-stack-exchange-api>. 2019.
- [81] Hutchings, B. L., Franklin, R. & Carver, D. “Assisting network intrusion detection with reconfigurable hardware”. *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*. IEEE. 2002, pp. 111–120.

- [82] *Improve regex to avoid backtrack and to use non-capturing groups*. <https://github.com/SonarSource/sonar-css/pull/110/>.
- [83] Inozemtseva, L. & Holmes, R. “Coverage is Not Strongly Correlated with Test Suite Effectiveness”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 435–445.
- [84] *Google Java Style Guide*. <https://google.github.io/styleguide/javaguide.html>. 2021.
- [85] *unit testing - What is the convention for JavaScript test files? - Stack Overflow*. <https://stackoverflow.com/questions/49632743/what-is-the-convention-for-javascript-test-files>. 2021.
- [86] Just, R., Jalali, D. & Ernst, M. D. “Defects4J: A database of existing faults to enable controlled testing studies for Java programs”. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 437–440.
- [87] Kabinna, S. et al. “Logging library migrations: A case study for the apache software foundation projects”. *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, pp. 154–164.
- [88] Kalliamvakou, E. et al. “The promises and perils of mining GitHub”. *Proceedings of the 11th working conference on mining software repositories*. 2014, pp. 92–101.
- [89] Kapur, P., Cossette, B. & Walker, R. J. “Refactoring references for library migration”. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010, pp. 726–738.
- [90] Kathryn A. Hargreaves, K. B. *Common Operators*. [http://web.mit.edu/gnu/doc/html/regex\\_3.html](http://web.mit.edu/gnu/doc/html/regex_3.html). 1992.
- [91] Kechagia, M. et al. “Effective and efficient API misuse detection via exception propagation and search-based testing”. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 192–203.
- [92] Khomh, F., Di Penta, M. & Gueheneuc, Y.-G. “An exploratory study of the impact of code smells on software change-proneness”. *2009 16th Working Conference on Reverse Engineering*. IEEE. 2009, pp. 75–84.
- [93] Kiezun, A. et al. “HAMPI: a solver for string constraints”. *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 105–116.

- [94] Kiezun, A. et al. “HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars”. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **21.4** (2012), p. 25.
- [95] Kim, M., Cai, D. & Kim, S. “An empirical investigation into the role of API-level refactorings during software evolution”. *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 151–160.
- [96] King, J. C. “Symbolic execution and program testing”. *Communications of the ACM* **19.7** (1976), pp. 385–394.
- [97] Kirrage, J., Rathnayake, A. & Thielecke, H. “Static analysis for regular expression denial-of-service attacks”. *International Conference on Network and System Security*. Springer, 2013, pp. 135–148.
- [98] Knab, P., Pinzger, M. & Bernstein, A. “Predicting Defect Densities in Source Code Files with Decision Tree Learners”. *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR ’06. Shanghai, China: ACM, 2006, pp. 119–125.
- [99] Ko, D. et al. “Api document quality for resolving deprecated apis”. *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 2. IEEE. 2014, pp. 27–30.
- [100] Kochhar, P. S. et al. “Adoption of software testing in open source projects—A preliminary study on 50,000 projects”. *2013 17th european conference on software maintenance and reengineering*. IEEE. 2013, pp. 353–356.
- [101] Langdon, W. B. & Harrison, A. P. “Evolving Regular Expressions for GeneChip Probe Performance Prediction”. *Parallel Problem Solving from Nature – PPSN X*. Ed. by Rudolph, G. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1061–1070.
- [102] Larson, E. & Kirk, A. “Generating evil test strings for regular expressions”. *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2016, pp. 309–319.
- [103] Le, X. B. D., Lo, D. & Le Goues, C. “History Driven Program Repair”. *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE, 2016, pp. 213–224.
- [104] Lee, J. et al. “A High Performance NIDS Using FPGA-Based Regular Expression Matching”. *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC ’07. Seoul, Korea: Association for Computing Machinery, 2007, pp. 1187–1191.

- [105] Lee, S. Y., Low, W. L. & Wong, P. Y. “Learning fingerprints for a database intrusion detection system”. *European Symposium on Research in Computer Security*. Springer, 2002, pp. 264–279.
- [106] Lesk, M. E. & Schmidt, E. *Lex: A lexical analyzer generator*. 1975.
- [107] Levenshtein, V. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. *Soviet Physics Doklady* **10** (1966), p. 707.
- [108] Li, N., Praphamontripong, U. & Offutt, J. “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage”. *Software Testing, Verification and Validation Workshops, 2009. ICSTW’09. International Conference on*. IEEE. 2009, pp. 220–229.
- [109] Li, N. et al. “Reggae: Automated test generation for programs using complex regular expressions”. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2009, pp. 515–519.
- [110] Li, Y. et al. “Regular expression learning for information extraction”. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2008, pp. 21–30.
- [111] Livshits, B., Whaley, J. & Lam, M. S. “Reflection Analysis for Java”. *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS’05. Berlin, Heidelberg: Springer-Verlag, 2005.
- [112] Lou, Y. et al. “Understanding Build Issue Resolution in Practice: Symptoms and Fix Patterns”. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 617–628.
- [113] Lu, J. et al. “Understanding Node Change Bugs for Distributed Systems”. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 399–410.
- [114] Lu, S. et al. “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”. *ACM SIGARCH Computer Architecture News*. Vol. 36. 1. ACM. 2008, pp. 329–339.
- [115] Ma, W. et al. “How do developers fix cross-project correlated bugs? a case study on the GitHub scientific Python ecosystem”. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 381–392.

- [116] Maalej, W. & Nabil, H. “Bug report, feature request, or simply praise? on automatically classifying app reviews”. *2015 IEEE 23rd international requirements engineering conference (RE)*. IEEE. 2015, pp. 116–125.
- [117] Madeiral, F. et al. “Bears: An extensible Java bug benchmark for automatic program repair studies”. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 468–478.
- [118] Majumder, S. et al. “Why Software Projects need Heroes (Lessons Learned from 1100+ Projects)”. *arXiv preprint arXiv:1904.09954* (2019).
- [119] Malaiya, Y. K. et al. “Software reliability growth with test coverage”. *IEEE Transactions on Reliability* **51.4** (2002), pp. 420–426.
- [120] *Mann-Whitney-Wilcoxon Test | R Tutorial*. <http://www.r-tutor.com/elementary-statistics/non-parametric-methods/mann-whitney-wilcoxon-test>. 2020.
- [121] Marsavina, C., Romano, D. & Zaidman, A. “Studying fine-grained co-evolution patterns of production and test code”. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2014, pp. 195–204.
- [122] Mcilroy, M. D. “Enumerating the Strings of Regular Languages”. *J. Funct. Program.* **14.5** (2004), pp. 503–518.
- [123] Meneely, A. et al. “Predicting Failures with Developer Networks and Social Network Analysis”. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT ’08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 13–23.
- [124] Michael, L. G. et al. “Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions”. *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’19. San Diego, California: IEEE Press, 2019, pp. 415–426.
- [125] Mileva, Y. M., Dallmeier, V. & Zeller, A. “Mining API popularity”. *International Academic and Industrial Conference on Practice and Research Techniques*. Springer. 2010, pp. 173–180.
- [126] Mileva, Y. M. et al. “Mining trends of library usage”. *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. 2009, pp. 57–62.

- [127] Mitra, A., Najjar, W. & Bhuyan, L. “Compiling PCRE to FPGA for Accelerating SNORT IDS”. *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ANCS '07. Orlando, Florida, USA: Association for Computing Machinery, 2007, pp. 127–136.
- [128] Moha, N. et al. “Decor: A method for the specification and detection of code and design smells”. *IEEE Transactions on Software Engineering* **36.1** (2009), pp. 20–36.
- [129] Moller, A. *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. <http://www.brics.dk/automaton/>. 2017.
- [130] Mozilla. <https://github.com/mozilla>. 2020.
- [131] MSDN - Matching Behavior. <https://msdn.microsoft.com/en-us/library/0yzc2yb0.aspx>. 2015.
- [132] Munaiah, N. et al. “Curating GitHub for engineered software projects”. *Empirical Software Engineering* **22.6** (2017), pp. 3219–3253.
- [133] Munson, J. C. & Elbaum, S. G. “Code churn: A measure for estimating the impact of code change”. *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE. 1998, pp. 24–31.
- [134] Nagappan, N. & Ball, T. “Use of Relative Code Churn Measures to Predict System Defect Density”. *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 284–292.
- [135] Neamtiu, I., Foster, J. S. & Hicks, M. “Understanding source code evolution using abstract syntax tree matching”. *ACM SIGSOFT Software Engineering Notes* **30.4** (2005), pp. 1–5.
- [136] *The Bro Network Security Monitor*. <https://www.bro.org/>. 2015.
- [137] O’Neill, M. & Ryan, C. “Grammatical evolution”. *IEEE Transactions on Evolutionary Computation* **5.4** (2001), pp. 349–358.
- [138] Ohira, M., Yoshiyuki, H. & Yamatani, Y. “A case study on the misclassification of software performance issues in an issue tracking system”. *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*. IEEE. 2016, pp. 1–6.
- [139] Omar, C. et al. “Active Code Completion”. *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 859–869.

- [140] *Online regex tester, debugger with highlighting for PHP, PCRE, Python, Golang and JavaScript*. <https://regex101.com/>.
- [141] *Outage Postmortem - July 20, 2016*. <http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016>.
- [142] Pacheco, C. & Ernst, M. D. “Randoop: feedback-directed random testing for Java”. *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM. 2007, pp. 815–816.
- [143] Palomba, F. et al. “Detecting bad smells in source code using change history information”. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2013, pp. 268–278.
- [144] Parnin, C., Bird, C. & Murphy-Hill, E. “Adoption and Use of Java Generics”. *Empirical Softw. Engg.* **18.6** (2013).
- [145] *Pearson’s moment coefficient of skewness | A Blog on Probability and Statistics*. <https://probabilityandstats.wordpress.com/tag/pearsons-moment-coefficient-of-skewness/>. 2015.
- [146] Pei, J., Han, J. & Wang, W. “Mining sequential patterns with constraints in large databases”. *Proceedings of the eleventh international conference on Information and knowledge management*. 2002, pp. 18–25.
- [147] *Performance report (Search)*. <https://support.google.com/webmasters/answer/7576553>. 2019.
- [148] Perkins, J. H. “Automatically generating refactorings to support API evolution”. *proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 2005, pp. 111–114.
- [149] Pham, R. et al. “Creating a shared understanding of testing culture on a social coding site”. *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 112–121.
- [150] Pingclasai, N., Hata, H. & Matsumoto, K.-i. “Classifying bug reports to bugs and other requests using topic modeling”. *2013 20Th asia-pacific software engineering conference (APSEC)*. Vol. 2. IEEE. 2013, pp. 13–18.
- [151] Pinto, L. S., Sinha, S. & Orso, A. “Understanding myths and realities of test-suite evolution”. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, pp. 1–11.

- [152] Piwowarski, P., Ohba, M. & Caruso, J. “Coverage measurement experience during function test”. *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press. 1993, pp. 287–301.
- [153] *PyGithub - PyGithub 1.45 documentation*. <https://pygithub.readthedocs.io/en/latest/>. 2020.
- [154] *pytest - Naming conventions and test discovery*. [https://docs.pytest.org/en/reorganize-docs/new-docs/user/naming\\_conventions.html](https://docs.pytest.org/en/reorganize-docs/new-docs/user/naming_conventions.html). 2015.
- [155] Herczeg, Z. *Performance comparison of regular expression engines*. [https://zherczeg.github.io/sljit/regex\\_perf.html](https://zherczeg.github.io/sljit/regex_perf.html). 2015.
- [156] Li, H. *Benchmark of Regex Libraries*. <http://lh3lh3.users.sourceforge.net/reb.shtml>. 2010.
- [157] Juárez, M. *Languages Regex Benchmark*. <https://github.com/mariomka/regex-benchmark>. 2020.
- [158] ADVENT, J. *JAVA REGULAR EXPRESSION LIBRARY BENCHMARKS – 2015*. <https://www.javaadvent.com/2015/12/java-regular-expression-library-benchmarks-2015.html>. 2015.
- [159] Maddock, J. *Regular Expression Performance Comparison*. [https://www.boost.org/doc/libs/1\\_41\\_0/libs/regex/doc/gcc-performance.html](https://www.boost.org/doc/libs/1_41_0/libs/regex/doc/gcc-performance.html). 2020.
- [160] *Regex use vs. Regex abuse*. <https://blog.codinghorror.com/regex-use-vs-regex-abuse>. 2005.
- [161] *c# - \d less efficient than [0-9] - Stack Overflow*. <https://stackoverflow.com/questions/16621738/d-less-efficient-than-0-9>. 2021.
- [162] *Perl-compatible regular expression optimizer*. <https://bisqwit.iki.fi/source/regexopt.html>. 2021.
- [163] *RegExr: Learn, Build, and Test RegEx*. <https://regexr.com/>.
- [Reg21] *regexp-tree - npm*. <https://www.npmjs.com/package/regexp-tree>. 2021.
- [164] *Regular expression visualizer using railroad diagrams*. <https://regexper.com/>.
- [165] *Replacing a Complex Regular Expression with a Simple Parser*. <https://www.honeybadger.io/blog/replacing-regular-expressions-with-parsers/>. 2017.

- [166] *Reporting-view filters*. [https://support.google.com/analytics/topic/1032939?hl=en&ref\\_topic=1726911](https://support.google.com/analytics/topic/1032939?hl=en&ref_topic=1726911). 2019.
- [167] *Rex @ rise4fun from Microsoft*. <http://rise4fun.com/rex/>.
- [168] Richards, G. et al. “An Analysis of the Dynamic Behavior of JavaScript Programs”. *SIGPLAN Not.* **45.6** (2010).
- [169] Rot, J., Bonsangue, M. & Rutten, J. “Proving Language Inclusion and Equivalence by Coinduction”. *Inf. Comput.* **246.C** (2016), pp. 62–76.
- [170] Roy, M. M. R. C. K. “An Insight into the Pull Requests of GitHub”. *Proc. MSR*. Vol. 14. 2014.
- [171] Saha, R. K. et al. “Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs”. *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 10–13.
- [172] Selakovic, M. & Pradel, M. “Performance Issues and Optimizations in JavaScript: An Empirical Study”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 61–72.
- [173] Sharma, T., Fragkoulis, M. & Spinellis, D. “Does your configuration code smell?” *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, pp. 189–200.
- [174] Shen, Y. et al. “ReScue: crafting regular expression DoS attacks”. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 225–235.
- [175] Shi, L. et al. “An empirical study on evolution of API documentation”. *International Conference on Fundamental Approaches To Software Engineering*. Springer. 2011, pp. 416–431.
- [176] Shin, Y. et al. “Evaluating Complexity, Code Churn, and Developer Activity Metrics As Indicators of Software Vulnerabilities”. *IEEE Trans. Softw. Eng.* **37.6** (2011), pp. 772–787.
- [177] Sipser, M. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston, 2006.
- [178] *Skewness | R Tutorial*. <http://www.r-tutor.com/elementary-statistics/numerical-measures/skewness>.

- [179] Spishak, E., Dietl, W. & Ernst, M. D. “A Type System for Regular Expressions”. *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. FTfJP '12. Beijing, China: ACM, 2012, pp. 20–26.
- [180] Staicu, C.-A. & Pradel, M. “Freezing the web: A study of redos vulnerabilities in javascript-based web servers”. *27th USENIX Security Symposium (USENIX Security 18)*(Baltimore, MD, 2018), USENIX Association. 2017.
- [181] Staicu, C.-A. & Pradel, M. “Freezing the web: A study of redos vulnerabilities in javascript-based web servers”. *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 361–376.
- [182] Stolee, K. T. & Elbaum, S. “Identification, impact, and refactoring of smells in pipe-like web mashups”. *IEEE Transactions on Software Engineering* **39**.12 (2013), pp. 1654–1679.
- [183] Stolee, K. T. & Elbaum, S. “Refactoring pipe-like mashups for end-user programmers”. *International Conference on Software Engineering*. Waikiki, Honolulu, HI, USA, 2011.
- [184] Tan, L. et al. “Bug characteristics in open source software”. *Empirical Software Engineering* **19**.6 (2014), pp. 1665–1705.
- [185] Teyton, C., Falleri, J.-R. & Blanc, X. “Mining library migration graphs”. *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 289–298.
- [186] *The Apache Software Foundation*. <https://github.com/apache>. 2020.
- [187] Thung, F. et al. “Automated Deprecated-API Usage Update for Android Apps: How Far Are We?” *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 602–611.
- [188] Tillmann, N., Halleux, J. de & Xie, T. “Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger”. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. New York, NY, USA: ACM, 2014.
- [189] Tillmann, N., Halleux, J. de & Xie, T. “Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger”. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 385–396.
- [190] Trasarti, R., Bonchi, F. & Goethals, B. “Sequence mining automata: A new technique for mining frequent sequences under regular expressions”. *2008 Eighth IEEE International Conference on Data Mining*. IEEE. 2008, pp. 1061–1066.

- [191] Trinh, M.-T., Chu, D.-H. & Jaffar, J. “S3: A Symbolic String Solver for Vulnerability Detection in Web Applications”. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1232–1243.
- [192] Tufano, M. et al. “When and why your code starts to smell bad”. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 403–414.
- [193] *Understanding the search syntax*. <https://docs.github.com/en/github/searching-for-information-on-github/getting-started-with-searching-on-github/understanding-the-search-syntax>. 2021.
- [194] Vahabzadeh, A., Fard, A. M. & Mesbah, A. “An empirical study of bugs in test code”. *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2015, pp. 101–110.
- [195] Veanes, M., De Halleux, P. & Tillmann, N. “Rex: Symbolic regular expression explorer”. *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE. 2010, pp. 498–507.
- [196] Wan, Z. et al. “Bug characteristics in blockchain systems: a large-scale empirical study”. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 413–424.
- [197] Wang, H. et al. “Clustering by Pattern Similarity in Large Data Sets”. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’02. Madison, Wisconsin: ACM, 2002, pp. 394–405.
- [198] Wang, P., Gina, R. & Stolee, K. T. “Exploring Regular Expression Evolution”. *Software Analysis, Evolution and Reengineering (SANER), 2019 IEEE International Conference on*. IEEE, 2019, pp. 502–513.
- [199] Wang, P. & Stolee, K. T. “How Well Are Regular Expressions Tested in the Wild?” *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 668–678.
- [200] Wang, P. et al. “An Empirical Study on Regular Expression Bugs”. *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 103–113.

- [201] Wang, P. et al. *Demystifying Regular Expression Bugs: A comprehensive study on regular expression bug causes, fixes, and testing*. 2021. arXiv: 2104.09693 [cs.SE].
- [202] Wang, X. et al. “Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs”. *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 631–648.
- [203] Wassermann, G. et al. “Dynamic test input generation for web applications”. *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 249–260.
- [204] *When you should NOT use Regular Expressions?* <https://softwareengineering.stackexchange.com/questions/113237/when-you-should-not-use-regular-expressions>. 2011.
- [205] *Why does this Javascript match() function Crash My Browser?* <https://stackoverflow.com/questions/12825950/why-does-this-javascript-match-function-crash-my-browser>. 2017.
- [206] Widyasari, R. et al. “BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies”. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1556–1560.
- [207] Wu, L. et al. “Transforming code with compositional mappings for API-library switching”. *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. IEEE. 2015, pp. 316–325.
- [208] Wüstholtz, V. et al. “Static detection of DoS vulnerabilities in programs that use regular expressions”. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017, pp. 3–20.
- [209] Xie, T. et al. “Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution.” *TACAS*. Vol. 3440. Springer. 2005, pp. 365–381.
- [210] Yang, Y.-H. E. & Prasanna, V. K. “Space-time tradeoff in regular expression matching with semi-deterministic finite automata”. *2011 Proceedings IEEE INFOCOM*. IEEE. 2011, pp. 1853–1861.
- [211] Yeole, A. S. & Meshram, B. B. “Analysis of Different Technique for Detection of SQL Injection”. *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. ICWET '11. Mumbai, Maharashtra, India: ACM, 2011, pp. 963–966.

- [212] Yeole, A. & Meshram, B. “Analysis of different technique for detection of SQL injection”. *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. ACM, 2011, pp. 963–966.
- [213] Yin, Z. et al. “How do fixes become bugs?” *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 26–36.
- [214] Yu, S. et al. “NBSL: A Supervised Classification Model of Pull Request in Github”. *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–6.
- [215] Zaidman, A. et al. “Mining software repositories to study co-evolution of production & test code”. *2008 1st international conference on software testing, verification, and validation*. IEEE. 2008, pp. 220–229.
- [216] Zhang, S. et al. “Combined static and dynamic automated test generation”. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 353–363.
- [217] Zhang, Y. et al. “An empirical study on TensorFlow program bugs”. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM. 2018, pp. 129–140.
- [218] Zhang, Z. et al. “Unveiling the Mystery of API Evolution in Deep Learning Frameworks A Case Study of Tensorflow 2”. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2021, pp. 238–247.
- [219] Zhao, H. et al. “Fully automatic wrapper generation for search engines”. *Proceedings of the 14th international conference on World Wide Web*. ACM. 2005, pp. 66–75.
- [220] Zhong, H. & Su, Z. “An empirical study on real bug fixes”. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 913–923.
- [221] Zhong, H. et al. “MAPO: Mining and recommending API usage patterns”. *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 318–343.
- [222] Zhu, H., Hall, P. A. & May, J. H. “Software unit test coverage and adequacy”. *Acm computing surveys (csur)* **29.4** (1997), pp. 366–427.