Exploring the Process and Challenges of Programming with Regular Expressions

Louis G. Michael IV

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Francisco Servant, Chair
Dongyoon Lee
Godmar Back

May 15, 2019
Blacksburg, Virginia

Keywords: regex, developer process, qualitative research, ReDoS

Exploring the Process and Challenges of Programming with Regular Expressions

Louis G. Michael IV

(ABSTRACT)

Regular expressions (regexes) are a powerful mechanism for solving string-matching problems and are supported by all modern programming languages. While regular expressions are highly expressive, they are also often perceived to be highly complex and hard to read. While existing studies have focused on improving the readability of regular expressions, little is known about any other difficulties that developers face when programming with regular expressions. In this paper, we aim to provide a deeper understanding of the process of programming regular expressions by studying: (1) how developers make decisions through the process, (2) what difficulties they face, and (3) how aware they are about serious risks involved in programming regexes. We surveyed 158 professional developers from a diversity of backgrounds, and we conducted a series of interviews to learn more details about the difficulties and solutions that participants face in this process. This mixed methods approach revealed that some of the difficulties of regexes come in the shape of: inability to effectively search for them; fully validate them; and document them. Developers also reported cascading impacts of poor readability, lack of universal portability, and struggling with overall problem comprehension. The majority of our studied developers were unaware of critical security risks that can occur when using regexes, and those that were aware of potential problems felt that they lacked the ability to identify problematic regexes. Our findings provide multiple implications for future work, including development of semantic regex search engines for regex reuse, and improved input generators for regex validation.

Exploring the Process and Challenges of Programming with Regular Expressions

Louis G. Michael IV

(GENERAL AUDIENCE ABSTRACT)

Regular expressions (regexes) are a method to search for a set of matching text. They are easily understood as a way to flexibly search beyond exact matching and are frequently seen in the capacity of the find functionality of ctrl-f. Regexes are also very common in source code for a range of tasks including form validation, where a program needs to confirm that a user provided information that conformed to a specific structure, such as an email address. Despite being a widely supported programming feature, little is known about how developers go about creating regexes or what they struggle with when doing so. To gain a better understanding of how regexes are created and reused, we surveyed 158 professional developers from a diversity of backgrounds and experience levels about their processes and perceptions about regexes. As a followup to the survey we conducted a series of interviews focusing on the challenges faced by developers when tackling problems for which they felt that a regex was worth using. Through the combination of these studies, we developed a detailed understanding of how professional developers create regexes as well as many of the struggles that they face when doing so. These challenges come in the form of the inability to effectively search for, fully validate, and document regexes, as well as the cascading impacts of poor readability, lack of universal portability, and overall problem comprehension.

Dedication

To my parents, for their love and support.

Acknowledgments

I would like to thank James Davis for both his guidance and for the blessing and the curse of interest in this topic as well as his immense assistance in editing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Regular expressions (regexes) are a text processing tool baked into all modern programming languages, as well as popular tools like text editors [53, 54]. Developers frequently incorporate regexes into their software. Estimates suggest that more than a third of JavaScript and Python projects include a regex [15, 20]. It is therefore surprising that we know so little about how developers interact with such a widely-used technology.

Existing investigations into regexes have focused on the source code and the technology rather than on the person using it. For example, other studies have delved into regex feature usage [15], regex evolution [58], and worst-case regex behavior [20]. But almost nothing is known about how software developers actually work with regexes. Without understanding what real developers think about regexes, how they go about creating them, what difficulties they face, and how they manage risks, we can only guess at what aspects can be improved.

We present the first large-scale qualitative study of the decision-making, difficulties, and risks developers face when they program with regexes. We surveyed 158 professional developers about their regex practices, and supported our findings by interviewing 11 professional developers to illuminate our findings. Through our investigation, we gained insight into developer perceptions and decision-making regarding regexes. We report on the decisions that developers make when working on regexes: to apply a regex or not; to write or reuse a regex; to identify test input; and whether the regex is correct. We look into the difficulties that developers feel when working through problems, ranging from mundane syntax problems

to complex reasoning about the relative importance of false positives and false negatives in the pattern-matching problem space. We learn about how developers work to handle these obstacles, the tactics they employ and the tools they use. We examine the risks of regex programming, and the surprising result that many developers are unaware of the portability and security problems associated with regexes.

Our core contributions are as follows:

- In order to provide as rich a glimpse of regex practices as possible, we conduct and analyze a large-scale survey of 158 professional software developers from a diversity of companies, including several major tech firms, to understand their regex practices. We furthered our findings by interviewing 11 developers.
- We synthesize this mixed-methods data to better understand the regex programming process: the decisions developers make, the difficulties developers face and how they handle them, and the degree of awareness around the risks of regex programming.
- We discuss the myriad implications of our findings, proposing a wide range of directions for future research, grounded in real-world practice.

1.1 Research Questions

In this study, we focus on understanding core human aspects of regex programming: how developers make their decisions, what difficulties they face, and whether they are aware of risks. Understanding these aspects of regex programming will motivate impactful new lines of research targeting the specific problems that professional software developers face. To this end, we study the following research questions:

- RQ1: Do developers reuse regexes?
- RQ2: From where do developers reuse regexes?
- RQ3: Do developers treat regexes like a lingua franca?
- RQ4: How do developers make decisions when programming regexes?
- RQ5: What do developers find difficult in programming regexes?
- RQ6: How do developers handle those difficulties in programming regexes?
- RQ7: Are developers aware of portability and security (ReDoS) risks when programming regexes?

To support our study of these research questions, we devised a general framework for the regex programming process, depicted in Figure 1.1. We adapted this framework from the general software engineering methodology of defining requirements, writing, validating, and deploying [41], and introduced a reuse stage based on our intuition that regexes are a kind of "function" complex enough to be reused like other software. We do not claim that our framework is exhaustive, but we believe it captures the crucial regex programming decisions that developers make. Using this framework, we were able to focus our methodology on developers' decision-making, challenges, and handling-mechanisms for each of the decisions in Figure 1.1.

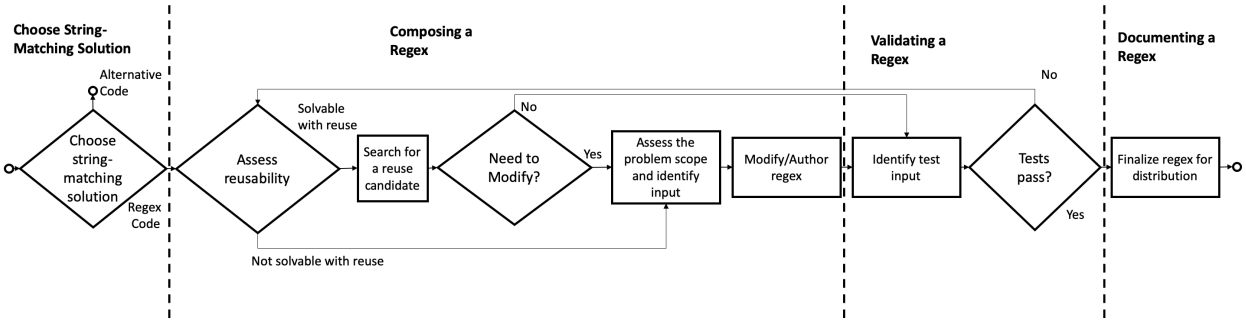

Figure 1.1: Generally the process of regex development broke down into four parts, with four major decision points (diamonds). This outline is what we used to frame the further investigation into developer decision making process and challenges.

Chapter 2

Literature Review

This chapter provides an overview of empirical regex studies. It also reviews related topics such as regex tools, engines and security vulnerabilities before providing some background on related software engineering topics which have yet to focus on regexes in specific but that we discuss as possibly interesting directions for regex research.

2.1 Empirical Regex Study

There have been several prior empirical evaluations on regexes, Hodován, Herczeg and Kiss [26] investigate the usage of regexes on the web to find that only 4% of all regexes on popular websites are unique. Chapman, Wang and Stolee [17] investigate whether given different ways to express the same semantics, one method is preferred over another. They performed a comparison by using Amazon mechanical Turk participants to complete a quiz on a regex function and evaluate based on the accuracy of the participants. From this, they observed that increased DFA size made a regex more readable among other slight preferences in syntax. Wang and Stolee [57] investigate how well regexes are tested and found that they are on average not tested very thoroughly when applying a notion of DFA coverage. Chapman and Stolee [16] investigate regex usage context and surveyed 18 professional developers all from one company on their usage of regexes. Wang, Bai, and Stolee [59] investigate if and how regexes change over time in open source Java code, introducing the idea of regex

evolution. They find that 95% of regexes don't change, and those that do tend to change by between 4 and 6 characters.

2.2 Regex Tools

Regexes have also been the topic of some tool development. Spishak, Dietl, and Ernst introduced a tool to statically detect issues typically producing run-time errors in regexes in Java [51]. Larson created two tools: one to use heuristics to check regexes for errors [29] and, joined by Kirk, an input generator to create potentially problematic input based on heuristics [30]. Other input generators were also been created by Arcaini, Gargantini, and Riccobene [5], based on mutating inputs, and by Veanes, Halleux, and Tillmann [56] with a C# tool, Rex, that produced input for testing and also provided support such as DFA generation.

2.3 Regex Engines

A regex engine is the implementation of the string matching algorithm that takes in a regex and input and evaluates them to find matches. There are two popular regex engine types. The dramatic distinction between them is their algorithmic complexity. The more widely used and more feature rich, but more algorithmically complex, of the two is the Spencer [50] backtracking approach. This approach is used in most programming languages, including Python and Perl, and searches for a match in a string by trying to match the first part of a regex and then marking points where it made a decision based on including or not including parts of input. It will then backtrack to these decision points if a match is not found, exhaustively trying all possible versions of a match. In contrast to this approach,

Thompson proposed a linear time regex matching approach in 1968 [55]. This algorithm has narrower feature support, including not supporting back references, a feature where you can define matching the same text again as was first referenced in a capture group. These two approaches are compared by Cox [18] in 2007 gaining popularity and traction for the Thompson approach. The Thompson algorithm is used in the RE2 [25] regex engine used in Go, and also inspired the Rust regex engine [46]. The algorithmic complexity of Spencer-style engines and features not only can lead to long evaluation times but also algorithmic complexity-style security vulnerabilities, as is discussed in the next section.

2.4 ReDoS

ReDoS is the only currently known major security threat that is introduced specifically as a result of leveraging regexes. Other security risks are possible, based on the correctness and context of a specific regex and how it manages data. ReDoS is a denial of service attack that exploits the super linear nature of Spencer-style regex engines. The attack works by providing malicious input to a regular expression, forcing it to consume large amounts of resources trying to discover a match. This starves the rest of the application of resources and can in turn result in a denial of service [11, 20, 45]. [21] investigates broader impacts of this style attack on NodeJS. Since Thompson-style engines guarantee linear evaluation time they are not subject to this attack. Tools have been developed to identify these style vulnerabilities through crafting malicious input for regexes [42, 49, 60, 62], which are used in creating attacks and identifying vulnerable regexes in the aforementioned studies.

2.5 Developer Perceptions, Practices and Information Needs of Other Technology

There have been a handful of studies that follow a similar format as this study using mixed methods approaches to shed light on software engineering practices. Notably Li, Ko and Zhu studying what developers think makes a great software engineer [32] and Bacchelli and Bird [6] investigating modern code review. Li, Ko and Zhu performed 59 semi-structured interviews where they talked to Microsoft developers about what they saw as the important traits of a great engineer. Bacchelli and Bird used a combination of tool usage data in conjunction with developer observation and interviews to surmise developer perceptions and practices surrounding modern code review. Thus far no one has studied regex programming in this mixed methods way.

Information gathering in software engineering studies pertains to how and what kind of information is needed to address a problem. This area of investigation has not been applied to regex programming but these studies employ approaches similar to the ones that we take in our study and provide recommendations for their respective communities. Breu *et al.* [13] investigated bug reports in Mozilla and Eclipse projects to analyze what factors impacted the speed and effectiveness of a report. They used this information to provide recommendations for bug tracking software. Buse and Zimmermann [14] surveyed 110 developers on when and how they used analytics in order to provide recommendations to analytics tools. Ko and Venolia [28] followed developers in 90 minute intervals to determine what information they required to be productive. They identified that developers frequently rely on their coworkers for information.

2.6 Code Search

Code search is the practice of searching through existing code sometimes with the goal of reusing the code that is located. Many studies have been conducted proposing tools for this end [7, 36, 39, 40, 44, 52]. Each of these studies proposed a variation on the approach optimized for a specific use case. Additionally Sadowski and Stolee [48] looked at developer practices around code search. Code search is one of the fields where regular expressions have been used frequently with tools such as grep being utilized or inspiring tools for this functionality but there has yet, to our knowledge, been a study specifically focused of search for regexes to re-use.

2.7 Code Reuse

Code reuse and copy paste behavior research looks at how, and how frequently, developers go or appear to go to other sources in order to find code to solve a problem they are facing. Wu *et al.* [61] analyses how developers utilize Stack Overflow, a popular question-and-answer platform, to write code. Mockus [37] investigates so called large-scale reuse at the file level and notes that it is fairly common to reuse whole files across different projects. Baltes and Diehl [8] empirically evaluates the frequency to which code from Stack Overflow is used with and without attribution in open-source projects. They find that up to half of developers will take code from Stack Overflow without properly attributing it. Fischer *et al.* [23] looks at Android code snippets on Stack Overflow for security vulnerabilities and analyzes the frequency which Android apps reuse security related Stack Overflow code, finding that almost all projects that contained reused security code contained at least one insecure reuse case. Similarly Zhang *et al.* [63] investigates the frequency of bugs in Stack Overflow sample

code, and finds that even highly rated posts will have API missusage. Overall this research has not looked specifically at copy paste behaviors of regexes from internet sources but, as previously discussed in [26], regex repetition, is a frequent phenomena.

Chapter 3

Methodology

We followed a mixed qualitative and quantitative approach [19]. We used a strictly qualitative approach to answer the broader more general research questions 4-6. Research questions 1-3 report this We emphasized a qualitative approach with the goal of *discovery*: to identify an exhaustive set of answers to our research questions, as well as understanding the details and context behind them.

3.1 Survey Methodology

3.1.1 Survey content.

We developed a 33-question survey with a mix of closed and open-ended questions. We asked participants about: (1) the process they follow when writing regexes; (2) their regex reuse practices; and (3) what awareness they have of regex security problems. We drafted our survey based on discussions with professional software developers, and followed best practices in survey design [27]. We refined the survey through internal iteration and two pilot rounds with graduate students. The final survey instrument is included for reference as appendix A.

3.1.2 Survey Deployment

After obtaining approval from our institution's ethics board, we took a two-pronged ethics board approach to surveying professional developers: acquaintances and strangers. First, we pursued snowball sampling by contacting developers of our acquaintance and asking them to take the survey and propagate it to their colleagues [12, 47]. Second, to increase the diversity of responses we posted the survey on popular Internet message boards frequented by software developers (HackerNews [1] and Reddit [2][1]). We compensated legitimate responses with a $5 Amazon gift card.

3.1.3 Filtering Results.

Posting our survey on Internet message boards introduced the threat of illegitimate responses. After the first 100 responses we performed a manual inspection and developed filters for validity. We only analyzed responses that were internally consistent[2], exceeded a minimum response time (5 minutes), and gave a thoughtful answer to at least one of our open-ended questions. This filtered out 253 responses, mostly from a spoofing campaign.

3.1.4 Demographics.

We received 158 responses from professional software developers. Our responses came from direct (51) and indirect (25) professional contacts, and Internet message boards (73), with no tracking information for 9 responses. The median respondent has 3-5 years of professional experience, works at a medium-size company, and claims intermediate regex skill.

[1] We posted to 3 subreddits: `r/SampleSize`, `r/coding`, and `r/compsci`.

[2] For example, their regex experience in a language could not exceed their total experience in that language. This inconsistency occurred in the automated survey responses.

We described increasingly complex regex features [24] for skill levels from novice to master. “Intermediate: For example, you have used more sophisticated features like non-greedy quantifiers (`/a+?/`) and character classes (`/\d|\w|[abc]|[^\d]/`).”(Figure 3.1) .

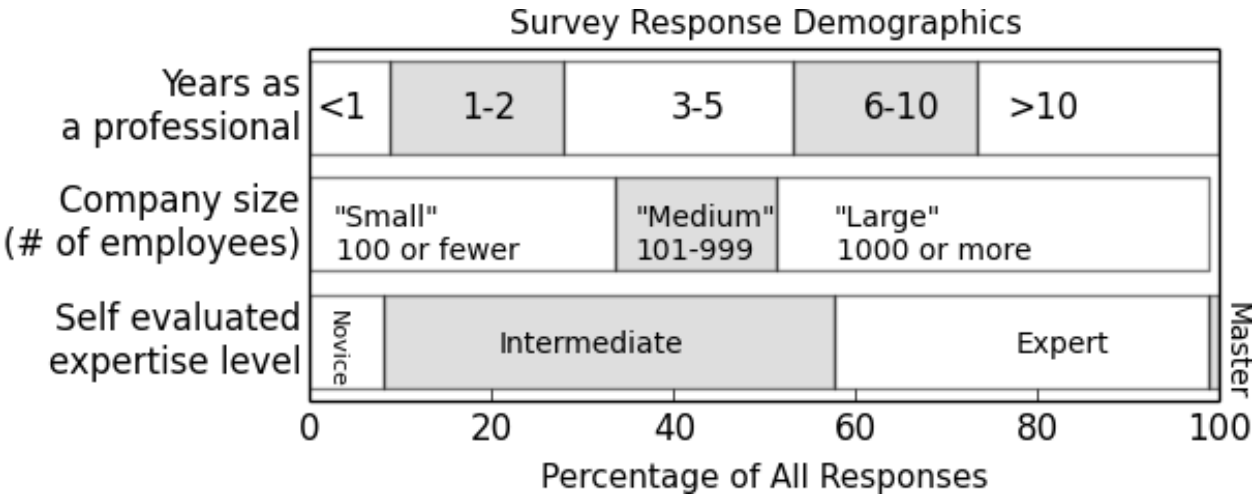

Figure 3.1: Our survey reached a diverse set of developers.

3.1.5 Survey Analysis

To analyze the free text responses provided in the survey, relevant free text responses were read on a question by question basis to develop a code book, they were then reread and coded. Following this the code book was provided to a college who then read and coded the responses. These two sets of codes were then reviewed to reach a consensus on the final code for each response.

3.2 Interview Methodology

3.2.1 Interview Procedure

The interviews were conducted in a semi structured-style, where an initial set of questions was prepared by the research team in advance of the interviews. These questions served as a guideline to highlight the process used by a developer when preparing regexes, but also to encourage noting difficulties that became apparent from analysis of the survey data. Our protocol was developed over several iterations within the team.

To find participants for the interviews all individuals who indicated being willing to participate in a follow up interview from the survey were contacted to request scheduling a follow-up interview. All individuals who responded were scheduled for an interview and upon completion of this interview were compensated with a $25 Amazon gift card. Each interview was conducted over Zoom [64], recorded, and this recording was provided to a third party transcription service. The survey procedure is provided as appendix B, and the consent form used prior to conducting the interviews is appendix C. In total 11 participants were interviewed.

Chapter 4

Results

4.1 Regex Reuse

4.1.1 RQ1: Do Developers Reuse Regexes?

When asked about their reuse habits regarding regexes, almost all (94%) of respondents indicated that they reuse regexes. 50% of respondents indicating that they reuse a regex at least half of the time that they use a regex (Figure 4.1 (a)). The most frequent reason to reuse a regex was to meet a common use case, *e.g.,* matching emails. This validates a previous hypothesis that developers may write regexes for a few common reasons [20].

Participants also mentioned time savings: "A good programmer doesn't re-invent the wheel."

4.1.2 RQ2: From Where Do Developers Reuse Regexes?

Developers most frequently said they reuse regexes from Stack Overflow, but they often reuse regexes from other code, whether their own, a colleague's, or miscellaneous code such as open source projects (Figure 4.1 (b)). About 90% of respondents reported re-using regexes from some Internet source, and about 90% reported re-using regexes from other code.

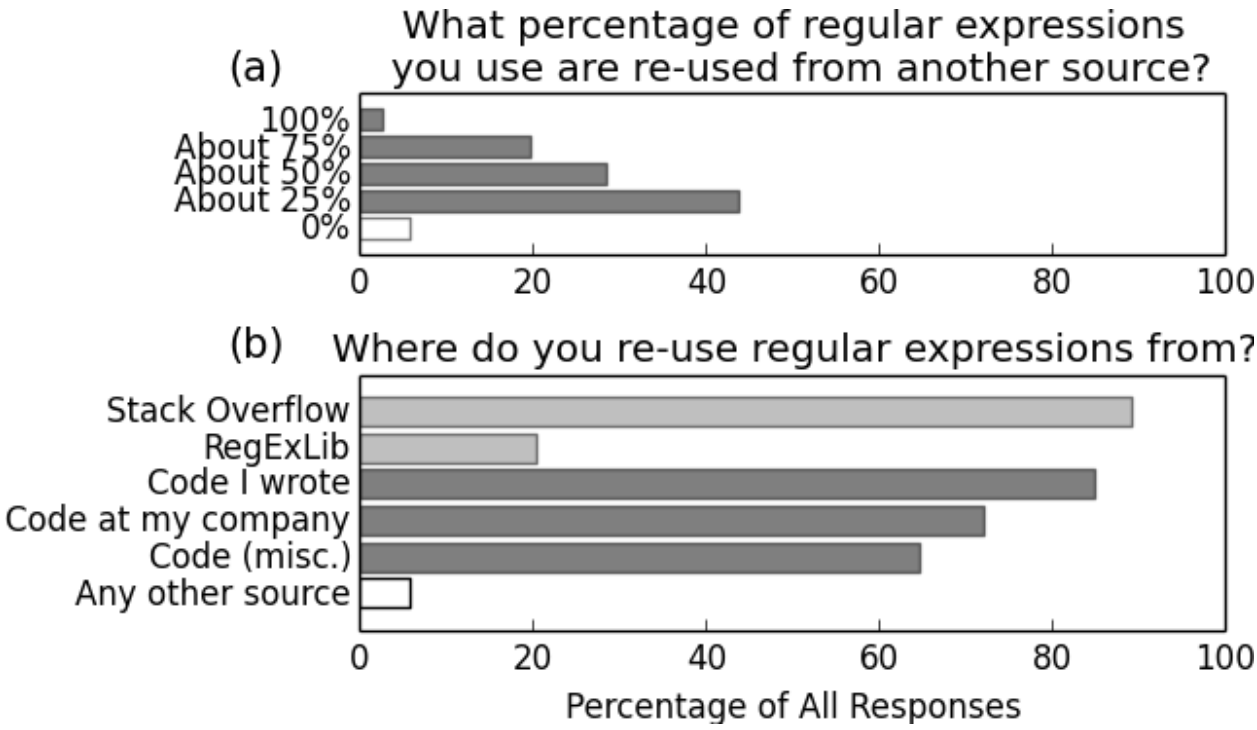

Figure 4.1: (a) When developers must use a regex, they frequently reuse them from another source. (b) Developers commonly reuse from the Internet and other code bases.

4.1.3 RQ3: Do Developers Treat Regexes Like a Lingua Franca?

We asked developers if their regex design process was influenced by language. Figure 4.2 (a) shows that 47% of our respondents do not have a design process that is language specific. Their actions match their beliefs: As shown in Figure 4.2 (b), respondents frequently reuse regexes without being confident that they were written in the same language. Only 21% of

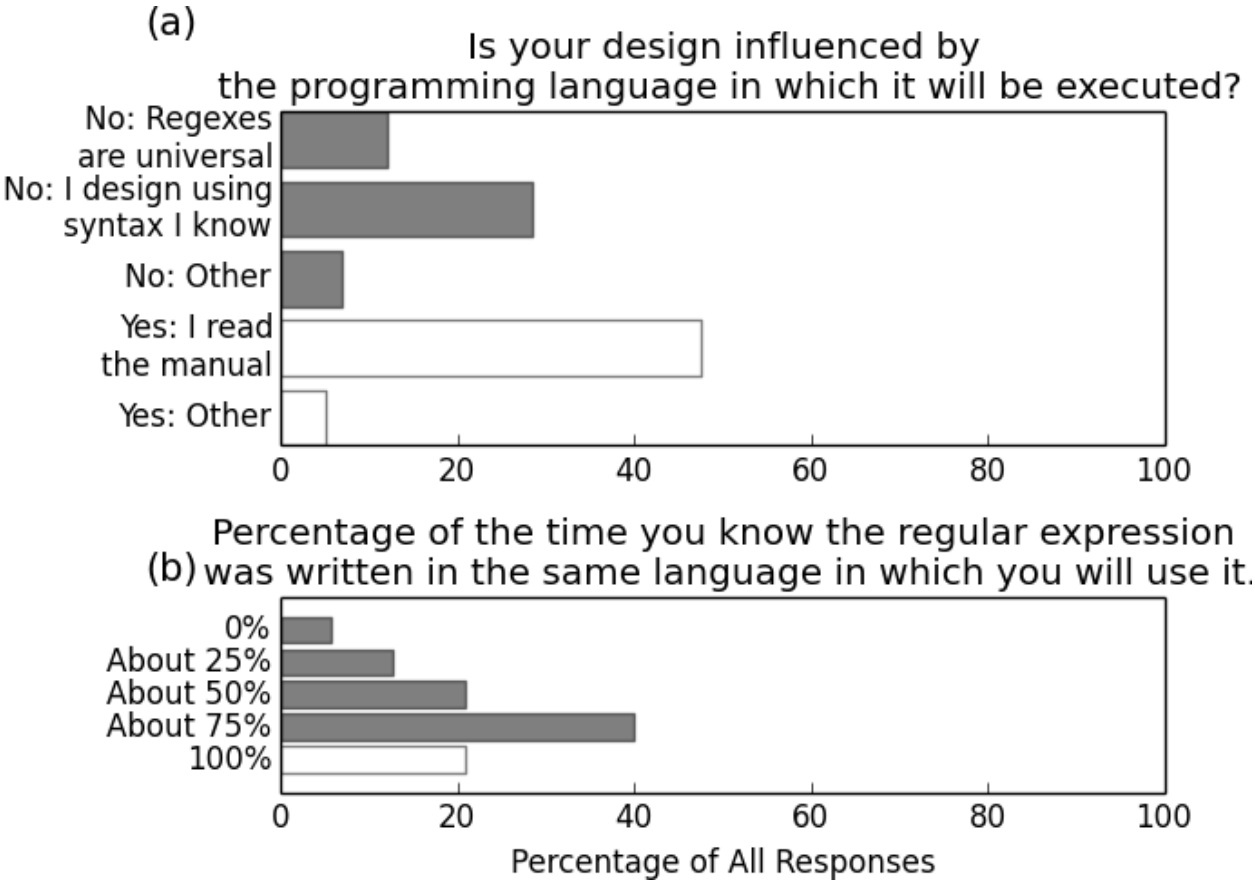

Figure 4.2: (a) Many developers design regexes without considering the programming language. (b) Developers' regex reuse decisions also imply belief in regex as a *lingua franca*.

respondents (34/158) were confident they never reused across language boundaries.

Table 4.1: Developer-Reported Decision Factors when Programming Regexes

Stage	**Decision**	**RQ2: Factors**
Choosing a Solution	Using regex vs. using non-regex code	Problem complexity, single choice, readability
Composing a Regex	Writing regex vs reusing regex	Problem commonality, reliability, time savings
	Which regex should I pick for reuse?	Best understanding, feature usage, length
	Match too much vs match too little?	Problem domain
Validating a Regex	Am I confident this regex is correct?	Choosing input or having sample data, result recipient
	Am I confident this reused regex is correct?	Trust of reuse
Documenting a Regex	How much documentation is required?	Personal opinion, complexity

4.2 RQ4: How do Developers Make Decisions when Programming Regexes?

One of the primary results of our work is an understanding of how developers make the decisions involved in programming regexes. We report it in the following subsections.

4.2.1 Assessing the Problem

We asked developers how they make two main decisions when assessing how to solve a string-matching problem: (1) writing a regular expression vs. writing alternative code, and (2) writing a regular expression vs. reusing an existing one.

4.2.2 Choosing a String-Matching Solution

We asked developers how they made the decision of which string matching solution to use, asking specifically about using a regex vs. writing alternative code.

Using Regex vs. Using Alternative Code

Developers reported making this decision based mostly on the **perceived complexity** of the problem. Developers perceive regexes as well suited for solving "Goldilocks" problems, neither too simple nor too complex. For simple problems, simple string APIs were preferred. As one interviewee said, *"if there's a string function that says the prefix should be this, I would prefer that over a regex …it's simpler to understand"* And when the problems are too complex, a survey respondent cautioned that regexes are also not a good solution: *"If a regex is complex enough that it's 'too complex' to write from scratch, it's probably also too complex a problem to solve with a regex."*

Another factor that developers considered when deciding to use a regex or its alternatives is their **perceived readability**. But developers disagreed on how readable regexes were, with some inclined towards and others away from using them. One participant said *"I stay away from tedious string parsing and splitting, and I see regular expressions as a tool to aid in conveying what you are trying to do"*, while other teams discourage regexes instead: *"With code review you want readable code and regexes are often not readable so they become less commonly used."*

Developers also mentioned that sometimes **a single choice is available**. For example, some built-in language and third-party APIs require developers to supply regexes to solve string matching problems. A common example is search tools: *"You find regular expressions and globs in search tools all over the place…in those cases, it's not really a choice."*

4.2.3 Composing a Regex

When developers decide to solve their string-matching problem with a regex, we asked them how they make two decisions while composing regexes. First, we asked how they decided

to write from scratch vs. reusing a regex. And when they opt to reuse, we asked them how they select reuse candidate(s). Developers also volunteered a decision that we had not initially considered, namely determining the relative merits of overly liberal or conservative matching (*i.e.,* too much or too little?). We updated our interview protocol to investigate this decision.

Writing Regex vs. Reusing Regex

Our participants often said they would reuse when they believed they were trying to solve a **common problem**. As one survey respondent said, *"If it's a common regex like various form fields I would reuse a regex, but for a more company/business use case specific requirement I would write a custom regex".*

Many respondents preferred to reuse regexes where possible to **improve reliability**, believing that regexes from a **trusted reuse source** would provide higher quality or better testing. For example, one interviewee said *"[A highly up voted Stack Overflow regex] is more likely to be right and account for edge cases"* The specifics of what constituted a trusted source varied. Some developers relied on a private team resource like a shared regex file, while others trusted highly up-voted Stack Overflow posts.

Another factor that developers considered when deciding whether to reuse was **saving time**, but they disagreed about the more efficient strategy. Some favored reuse (*"Similar to writing code, finding a working example and adapting it is faster"*), while others said that *"Writing from scratch often requires less time than searching for a suitable one."*

Which Regex Should I Pick for Reuse?

Developers who opt to reuse must often choose from multiple candidate regexes. Our participants said they preferred simpler regexes when given the choice, but measured complexity in different ways. One interviewee showed a preference for the **fewer special characters**, saying *"I just try and pick the one I have the most understanding of …the one with the fewest special characters."* Another emphasized **length**. *"[If one] answer is half the length I'm going to go with that one."*

Match too much vs. match too little?

We did not initially anticipate this decision, but added it to our interview protocol based on information volunteered in an early interview. When composing a regex, developers discussed needing to have an understanding of the context in which their regex would be used. In some situations they preferred their regex to be overly liberal, matching too much, while in others they wanted to be conservative, matching too little. They said *"what might be tricky is deciding whether or not you want to match it too much or match too little"*, and pointed to validation as a **particular context** where matching too little (false negatives) was preferable to matching too much: *"I'd much rather match too little than too much [to avoid introducing] garbage data]."*

4.2.4 Validating a Regex

We asked developers about two aspects of validation: how they decide whether a regex is correct, and whether their process changes when they are re-using a regex rather than writing their own.

Is This Regex Correct?

Our participants' confidence was often tied to having **comprehensive sample input** for their regex. *"I'm usually pretty confident about them…we have a pretty much an unlimited … sample pool of things I can use to test."* On the other hand, a participant described editing a colleague's old regex as difficult because they did not perfectly understand the input space.

In **non-customer-facing contexts**, some participants had a lower standard for correctness. They said things like *"I just kind of eyeball it …somebody …will probably test the edge cases,"* usually when the recipient of the data was a team member rather than a customer.

Is this Reused Regex Correct?

Our participants were split on whether and how to change their validation strategy for reused regexes. Some developers viewed reused regexes as better tested or more and as a result **did not work to validate as thoroughly** as when they wrote from scratch, saying things like *"I'll usually trust re-using an expression more …[and] skip [some validation phases]"*. But others treated reused regexes cautiously, *"I am aware of the security implications of using something from public sources."*

4.2.5 Documenting a Regex

How Much Documentation is Required?

Perhaps related to the differing opinions on the readability of regexes that we identified earlier, interviewees **disagreed** on the extent to which **regex documentation was required**. Many felt that a well-written regex would not require documentation: *"I would say that …most people would consider them self documenting,"* others thought that the amount of

Table 4.2: Developer-Reported Difficulties and Handling Methods in Programming Regexes.

Stage	RQ3: Difficulty	RQ4: Handling Mechanism for Difficulty
Cross-cutting	Understanding the Problem	Read data, break down problem
	Understanding the Regex	Using tool support, breaking down regexes, adding documentation
Composing	Searching for Reuse Candidates	Decomposing the regex, searching for similar code, personal regex library
	Unfamiliar Syntax	Using tool support reading the regex documentation
Validating a Regex	Testing Edge Cases	Test all available input
	Testing Enough Inputs	Work to find or generate more real cases, property testing

documentation depended on regex complexity, *e.g., "something that has two or three levels of parentheses …I will try to break them apart into smaller pieces with more comments."*

4.3 RQ5 & RQ6: What do Developers Find Difficult in Programming Regexes, and How do They Handle Those Difficulties?

Intertwined with their decision-making process, participants mentioned many difficulties during regex programming. As in the preceding section, we analyzed survey responses and then used our interview phase to probe for additional details about the challenges and ways developers handle them. For clarity, in this section we accompany each challenge with the way(s) participants described handling them[1]. Table 4.2 summarizes our findings. As indicated in Table 4.2, the first two challenges we discuss were cross-cutting, spanning several stages of the regex programming process. We then discuss two common challenges that our participants face in composing regexes, and two challenges they face in validating regexes.

[1]We chose our words carefully here. We refer to what participants described as "handling problems" because many participants were aware of limitations or gaps in their approaches. We discuss potential research directions to address some of their concerns in chapter 5.

4.3.1 Understanding the Problem

Difficulty

Multiple developers reported the difficulty of fully understanding the string problem that they are trying to solve, *e.g., "The most difficult thing with regular expressions tends to be defining the problem."* Developers mentioned this difficulty affecting many of the stages of programming with regexes. In particular, one participant mentioned understanding the problem as a difficult step in both writing and validating regexes with tests: *"Having clear understanding of the task …helps not only to write the tests but also to write/pick the regex."* This may be tied to the importance of a good set of inputs during the validation step, mentioned by other participants in §4.2.4.

Handling

Developers primarily handled the problem of problem definition by **studying sample inputs** for patterns. *"I tried to generalize what I'm looking at and [then] craft the regular expression."* When they could not understand the problem at a glance, developers discussed **breaking the problem into smaller pieces**. One interviewee described this as *"[getting] very methodical …much like I attack any programming problem, where I break it into manageable pieces."*

4.3.2 Understanding the Regex

Difficulty

Their opinions about regexes being "self-documenting" notwithstanding, many of our participants said they perceive regexes as difficult to understand. Several interviewees summed this up well, variously remarking that *"The syntax of regular expressions is kind of terse,"* and *"The regex syntax is cryptic,"* and simply *"Illegible gibberish."* Developers reported that **the difficulty of interpreting regex syntax as a pattern, *i.e.*, "reading regexes," impacted all stages of the regex programming process**. The difficulty of understanding regexes is exacerbated by frequent *"lack of comments/documentation, poor style,"* a point raised by many developers.

Handling

Developers reported two strategies to handle this difficulty: using tools to improve their own regex comprehension, and documenting regexes to improve comprehension for others. The most common tools mentioned by participants were visualization aids like graphical visualization and syntax highlighters. The most praised among these were the **built in syntax highlighting** for the JetBrains IDEs, *"Jetbrains has my back - IDE syntax highlighting."*

To improve regex comprehension for others, developers described their regex documentation strategies. One common method was to **break regexes across multiple lines**, providing a comment about each individual part of the regex. This is not a universally supported feature across all regex implementations, but was encouraged by participants when it was available. From a developer interview: *"Hopefully you're using a programming language where you could break it into multiple lines and comment those."*

In addition, some developers encourage others to document their regexes when they come across them in code review. In those cases, some developers push to include more detailed commenting, *e.g., "Put a plain language explanation in comments…Have as many examples of matching and unmatching text as is appropriate in the comments."*

4.3.3 Searching for Reuse Candidates

In this and the following subsection we introduce challenges specific to two stages of the regex programming process: searching for reuse candidates and validating regexes.

Difficulty

In order to reuse a regex, developers first need to search for a regex that is worth reusing. Multiple participants lamented the difficulty of this process, which mostly affects the regex composition stage.

When our participants search for a regex to reuse, they usually try to leverage general search tactics with mixed success. In particular, **developers find it difficult to frame their search in a way that is understood by existing search tools** — it is difficult to express their abstract string-matching problem as a plain-text query for a search engine. For some tasks the desired regex is easy to search, *e.g.,* "email regex", but developers often find it difficult to express the regex that they need. From a developer interview: *"It's hard to …query the problem you're trying to solve. Sometimes it's so domain specific."*

In this case, developers face both the difficulty of understanding the problem itself (§4.3.1) and the difficulty of articulating it to an existing search engines.

Handling

Developers reported three strategies to handle this difficulty.

Some developers choose to **decompose the regex** into smaller pieces that may be easier to search. This is expressed well by one of the survey respondents: *"If I can't find an existing regex that fits my need, I will start searching …[for] pieces that will help me construct the final regex."*

Another popular approach was an indirect search. Developers commonly **search for code** that may use a relevant regex. For example, this was described in an interview when the participant said: *"I would say intuition, but also sort of like code that's close by for sure. A file next to it or maybe it's an implementation of something that conforms in some interface."*

Finally, some participants maintain **personally curated lists for regex reuse** that they will consult. For example, one survey respondent mentioned that they would *"refer to the regex section of my personal notebook"* when searching for reuse candidates. This handling mechanism seems similar to having a personal library of utility functions that you copy into your codebase as needed.

4.3.4 Unfamiliar Syntax

Difficulty

Developers frequently discussed their lack of familiarity with the syntax as an obstacle, *e.g., "You have to remember what each symbol in that string means …they're non-intuitive."* Another developer expressed a reliance on regex-specific reference charts: *"I need a little cheat sheet that has to outline what each symbol does."*

Handling

Developers relied on two handling strategies to address difficulties with syntax.

Unsurprisingly, developers often mentioned referring to their programming language's **regex documentation** to support their composition of regexes, *e.g.,*: *"…If [searching for a regex to reuse] fails, I will start reading the regex documentation."*

But another common mechanism is the usage of **tool support**. *"Anytime I am curious about a regex [I] go to regex101.com… You type in your regex and some examples and it'll match or not match in real time and that's just useful."* Developers also appreciated tools that incorporated documentation. One participant noted that *"there's a bar on the side of [regexr.com] …[You can] click on every sort of regular expression piece of syntax and it'll show you an example."*

4.3.5 Testing Edge Cases

Difficulty

Developers also found it difficult to identify *corner cases* or *edge cases*, terms colloquially used to refer to *boundary values* [43]. They often expressed the *"Difficulty in [coming up with] corner case inputs and outputs"* and the fact that it is *"Tough to imagine all edge cases to test."* They found it hard to understand even well defined problems.

Handling

Some developers handle this problem by generating their own input, while others rely on real-world input data from others. For simple problems developers say they think through

the problem space and generate their own input, but noted that this approach has its limits. *"If the regular expression is …simple enough that thinking about the entire scope of the input space is feasible …It's really the case where it really grows into a massively complex one that [is problematic]."*

Other developers gained an understanding of their problem by reading data they had on hand, but they acknowledge that they then tend to **only address the edge cases that manifest in the available input data**. One participant remarked that *"[I] look for everything that I can get from production…that's my input …that's my unit test."* This participant went on to say *"[but] unfortunately the input that I get can't be 'universal'."*

4.3.6 Testing Enough Inputs

Difficulty

Many developers reported on the difficulty of validating regexes. Regexes are powerful and flexible tools, but in consequence can be very difficult to validate. A survey respondent summed up this idea well: *"an infinite regression problem, to test a regex …would require regexes."* In particular, developers find it difficult to come up with sets of sample inputs for testing that they would consider complete. This was a frequent complaint: *"insufficient sample inputs / unknown set of sample inputs."* Developers found it difficult to feel confident in testing common cases; possibly as a result of the problem being poorly defined.

Handling

Developers handled this through ad hoc approaches to expand their collection of inputs. One participant relied on the **expertise of other humans** to do this: *"Testing literally*

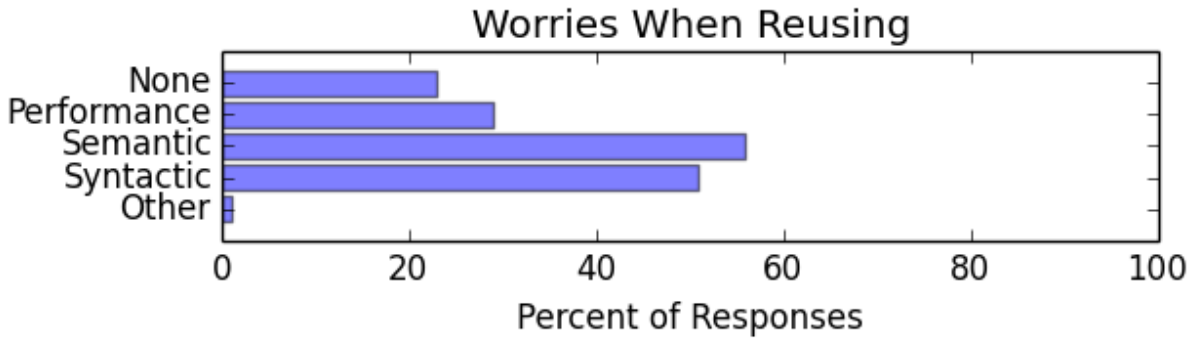

Figure 4.3: Developers responding to the second survey when asked about which they worried about when reusing expressed a range of concerns, emphasizing semantic portability issues, that a regex would not work as intended. Developers also worry less about performance issues, where a regex could slowdown overall execution time, but this may be attributed to lack of awareness of performance vulnerabilities as discussed in §4.4.2
.

every scenario is unfortunately not a realistic solution…working with the QA and the clients to get a diverse set of real world documents." Another participant said they automatically generated additional input using **property based tests** [34].

4.4 RQ7: Are Developers Aware of Portability and Security (ReDoS) Risks when Programming Regexes?

Developers encounter two risks when programming regexes: correctness concerns due to regex (non)-portability, and security concerns due to ReDoS. In this section we describe developers's awareness of these risks, as well as their handling mechanisms.

4.4.1 Portability Risks

Awareness

We asked developers whether they worry about a series of risks involved in regex reuse: syntactic, semantic, and performance differences when reusing regexes between languages. We report their answers in Figure 4.3.

Developers who responded to the second survey worry about syntactic and semantic differences (over 50%), and around a third (29%) also worry about performance differences. Our interview participants provided more details about specific portability problems they have resolved in the past. They reported re-factoring regexes with unsupported features which at times would blend semantic and syntactic differences, *e.g., "escape sequences …vary across systems"* and *"[Go regexes] don't support some of the constructs that are available in other languages."*

An interesting facet of Figure 4.3 is that a substantial portion of developers also reported not worrying about any of these reuse risks. Our interviews shed some light into why many developers do not worry about regex reuse risks: **many are not aware that reuse carries portability risks**.

Furthermore, other respondents described the shock they felt when they first learned of regex portability issues, *e.g., "I certainly didn't know [before that incident] …what the hell is that?"* Ignorance of this issue exposes developers to code correctness issues due to improper regex validation.[2] We note that these assumptions can affect code correctness whether or not regexes are being reused, simply as a result of an incorrect mental model for regex behavior.

[2]For example, consider the participants we described across several earlier sections, who reuse from a "trusted source" like Stack Overflow and do not validate carefully as a result.

Handling

Most developers handled missing feature support or blatant syntax differences by consulting language documentation and making a **translation**. For example, *"for syntactic differences, I look at a regex cheatsheet and find the appropriate syntax for my environment"*if they understand the need for translation, developers do not find this translation difficult, though it can be **frustrating**. In some cases the need for regex translation can even influence larger project decisions. One participant was considering migrating a project from one language to another, but decided against it: *"Transitioning our common regular expressions …kind of a headache."* Other developers would not go to the effort to make a translation. If they found that a regex reuse candidate would not work in their regex dialect, they would **start their reuse search process over** to find one that did not need a translation: *"Sometimes I ported it. Sometimes I went looking for another."*

The primary way that most developers discussed dealing with further concerns was **testing** to confirm the regex they were reusing behaved in the way they expected. *"I run the regex against various tests to ensure it outputs as expected."*

4.4.2 ReDoS

Awareness

Regex opens the somewhat obscure security concern of ReDoS. ReDoS is fairly avoidable. If proper steps are taken to sanitize input and to not use superlinear regexes, it is a nonissue. The first step to avoiding the issue, however, is being aware of the problem. When asked if they knew what ReDoS was, developers were overwhelmingly unaware. **Only 35% of all developers surveyed knew of the possible vulnerability.** This is concerning since

the vulnerability is easy to both introduce without noticing and to slip through validation without seeing an issue. We note that ReDoS is a vulnerability rather than an exploit, and is discussed as a risk since it is only relevant if the regex matches against malicious input. This is not always possible since not all regexes accept input in a way that can be defined by an attacker. This means that developers whose regexes process known or trusted input need not be concerned about it. Nevertheless, we were surprised that the majority of participants were unaware of ReDoS.

Handling

Beyond their (occasional) awareness of this security risk, developers currently do little to nothing to combat performance issues and feel ill-equipped to identify performance issues leading to ReDoS in their regexes. When discussing performance issues and challenges in interviews, many developers said they only worry about regexes that introduce noticeable *"lag"*: *"I just wait and see if it becomes an issue."* Only one interviewee referred to ReDoS (*"catastrophic backtracking"*) thanks to a related feature in regex101.com [4].

We emphasize that these developers hold a misconception about worst-case regex performance issues, which may not manifest on typical input but only on malicious input. The feature in regex101.com is not a ReDoS detector but rather a diagnosis tool suitable if the developer already knows about the worst-case input.

Part of these behaviors may be because developers **lack tools or knowledge** about solving regex performance problems. For example, Davis *et al.* reported that input sanitization is an easy mitigation for many ReDoS vulnerabilities [20], but none of our participants mentioned this approach. And when asked what is difficult about validation, one developer simply stated "Performance and security risks," implying that they do not know about the four

tools tailored to this problem [42, 49, 60, 62].

Chapter 5

Discussion and Implications

5.0.1 Recommendations for Regex Developers

Our findings have implications for many research directions.

5.0.2 Evaluating Regexes

Our findings show developers need to assess various qualities of regular expressions. Our findings for RQ2 and RQ4 explain how developers use various factors of **complexity** and **quality** to make decisions and overcome difficulties. For example, developers considered both regex complexity and quality as important factors to decide which regexes to reuse. Such estimations of regex complexity and quality are normally taken manually, by simply looking at the regex.

Regex Metrics

Our findings highlight the value of developing regex metrics to automatically measure the quality and complexity of regexes in a sense that will help developers make decisions when programming regexes. For source code, metrics already exist to capture some of its complexity and quality, *e.g.,* cyclomatic complexity [35]. We pose that regex metrics would have a strong impact in the productivity of developers when programming regexes, since most of

the decisions that they make consider some metric. We elaborate further on our envisioned usage of regex metrics in the following sections.

5.0.3 Automated Support for Reusing Regexes

Our findings highlight multiple difficulties in regex reuse, such as defining the problem for a search engine query and reusing across regex dialects. They also point out the characteristics that developers value when reusing and the interesting practice of keeping regex lists for reuse. These findings open multiple opportunities for research.

Semantic Regex Search

Currently, developers find it difficult to use search engines to look for regexes to reuse, since it is hard to express their string-matching problem in a few words — particularly considering that the problems themselves are hard to understand (see §4.3.1).

Developers would benefit from a search engine that allowed them to express string-matching problems in a domain-specific way, *i.e.,* semantic regex search. Such a semantic regex engine could be more useful to developers by taking inputs that are regex-specific. Some examples of these could be: (a) a list of inputs that match or non-match the regex that they need, (b) a regex that resembles the regex that they need (c) a code context in which the regex would be used. The approaches to processing these inputs would vary, but we pose that such a search engine would make it easier for developers to express the problem that they are trying to solve.

Regex Repositories

Since developers mentioned keeping their own lists of regexes for future reuse, it may be useful to empower them in that practice. Regex repositories would strongly complement semantic regex search, particularly if the stored regexes contain additional metadata, such as: the category of string-matching problem that they solve (possibly from Chapman and Stolee's [16] proposed classification of common regex solutions), portability or performance problems, or other indicators of quality via regex metrics or user ratings. One regex repository does currently exist, RegExLib [3], but it encompasses a very narrow set of facets in which to preform search, and is also relatively small given the relative size of other existing regex corpuses.

Metrics-based Regex Ranking

Regex metrics would complement semantic regex search by allowing the ranking of results according to various metrics. Developers expressed that in most cases they value in regexes to reuse low complexity — as short length, few features used —- and high quality — as coming from a reliable source and being better tested.

Regex Dialect Translation

Finally, developers also expressed the difficulty of reusing regexes that were created in a different dialect. Ideally, a regex search engine should also include mechanisms to understand the regex dialect for which a regex was created, as well as mechanisms to refactor it for the dialect in which it will be used.

5.0.4 Automated Support for Composing Regexes

Our findings point out difficulties with composing regexes, *e.g.,* dealing with difficult syntax that is hard to remember. We also found many qualities that developers value in regexes, *e.g.,* short length and reduced feature usage. These findings motivate many avenues for research.

Live Support for Regex Composition

Now that we better understand some of the characteristics that developers value in regexes (see §5.0.2), we could develop assistants to support developers in composing regexes with those desirable attributes. Such assistants could help developers to, *e.g.,* break down their regexes, use fewer advanced features, or decide between matching too match or too little.

Regex Refactoring

The same information about desirable qualities in regexes could be applied to develop regex refactorings that would improve the quality of regexes. Regex refactoring could be applied on demand or automatically over a whole code base. Refactoring is another concept that is highly valued in object oriented programming, and it could highly benefit regex programming too. Chapman and Stolee [16] already proposed the idea of regex refactoring. Our findings throw more light into what kinds of refactorings would be desirable for developers.

Automatic Regex Composition

Another interesting research direction would be to try to fully automate regex composition. Some existing work has made advances in this direction by composing regexes from examples

of matching and non-matching input [9, 10, 22, 33]. A different approach to automate regex composition may be by feeding the algorithm with partial regexes that solve pieces of the problem — since developers seem to be already decomposing the problem in their manual composition efforts.

5.0.5 Automated Support for Validating Regexes

Developers mentioned the difficulty of validating regexes, particularly in testing edge cases and in deciding that they tested enough input cases. They currently handle such difficulties manually, by testing all the input that they have available. These difficulties motivate research in at least two directions.

Regex Input Generation for Humans

Many tools have been proposed to generate input for testing regular expressions [5, 31, 38, 49, 56]. Surprisingly, none of our studied developers mentioned using these tools for input generation.

We realize that further study would be necessary to understand the extent to which these tools are adopted. However, we believe that further research is motivated to study what developers consider *good* or *relevant* input, as well as to evaluate regex input generators when their output is consumed and judged by humans.

Boundary Value Analysis for Regexes

Methods like boundary value analysis and equivalence partition help software developers define and test *edge cases* and are widely-known. Thus, further research is motivated to

adapt these techniques to regexes in particular or to develop other methods to support developers in defining boundary values for regexes.

5.0.6 Automated Support for Documenting Regexes

We observed that some developers felt that regexes should be self-documenting, others thought that documenting regexes is necessary, and others thought that the decision depends on the complexity of the regex. But, in addition, we found the things that developers value in regex documentation: breaking down the regex into pieces, documenting each piece, and the inclusion of both matching and non-matching input, as well as a plain description of what the regex does.

Automatic Regex Documentation

These findings motivate research to automatically document regexes, since many of the pieces of information that developers value could potentially be generated automatically. Machine learning techniques could be developed to automatically break down regexes by *learning* from examples. Also, regex input generation techniques (see §5.0.5) could be adapted to generating few inputs that would be highly relevant for humans to understand the regex, and that also covered matching and non-matching input.

5.0.7 Understanding Regexes

Many developers mentioned the *terse* and *cryptic* nature of the syntax of regexes, and how that makes it very difficult for them to understand them.

Novel Regex Syntax for Comprehension

We believe this widely-held sentiment calls for further research into new syntax for regexes that makes it easy for humans to understand. Since developers are already breaking down regexes in multiple lines, that may give space for a more verbose syntax that could be more easily understood.

Chapter 6

Threats to Validity

6.1 Construct Validity

In order to pose questions about the regex process to developers we first outlined a general process based on general software engineering practice. This limited the scope to which participants may have reflected on their process or challenges to the scope that we framed. Further participants are by no means guaranteed to have the same understanding of the framing or even of regex usage or context.

6.2 Internal Validity

The researchers on this study analyzed the data in a very human way by reading, summarizing, categorizing and discussing things said by other humans. This has the potential to introduce bias at many levels and also may limit repeatably of this study since other individuals may have interpreted the same information in different ways. This is a very natural limitation of this kind of qualitative research.

6.3 External Validity

This study investigated a very small portion of software engineers. In the aggregate survey for part of the sample we used snowball sampling starting with professional contacts of the team, biasing towards people known by the team and at companies the team already worked. This may then compound with the internal risks and exacerbate previously held beliefs of the team. Further it is possible that developers behaved in a specific way as a result of being aware that they were under observation.

Chapter 7

Future Work

There is room for continued research on the topic of developer perception, practices and challenges of regex. Several perspective to this topic were not fully explored due to time constraints in both our interviews and survey. One such aspect was tool usage, although this paper provides a series of recommendations for tool developers and discusses tool usage with developers, it does not perform a comprehensive overview of all existing tools nor does it observe developers interacting with any tools in specific. Another angle that this research does not fully explore is the process of developers collecting and curating examples. We discussed with developers some challenges related to creating regexes, and in this discussion, developers would allude to viewing sample data and finding patterns, as well as trying to ensure they had representative sample data. At times developers mentioned this step involving interfacing with colleagues, clients, or other external actors. This information-gathering step seemed crucial to producing a regex that functioned correctly in production but little is known about the details of this step. Further, it was not addressed in this work how regexes were similar or different to other built in programming features, other APIs or other tools such as git or SQL. While it is intuitive that regexes were harder for developers than a for-loop, it was not contrasted with how some of the phenomena and information needs discussed in this work may resemble other tools and technologies used by developers. Finally, a question raised by this work is whether a regex really is the best approach to string matching problems. Developers seem to be able to identify problems as best solved

by regexes but this may be a result of lacking other tools with enough power to solve such problems. It is possible that an additional level of abstraction or more powerful language built in string functions could solve some of the challenges developers face with regexes.

Chapter 8

Conclusions

Regexes are powerful tools that developers find valuable. Regexes can also hard to work with. We identify six difficulties developers face when using regexes and strategies they employ to deal with these difficulties. The way that developers cope with regexes leaves a great deal of room for improvement. Developers are also critically unaware of risks they take when using regexes with only 35% being aware of security vulnerabilities associated with their usage. We propose many lines of research and new ways to support developers when working with regexes.

Bibliography

[1] Hacker news. https://news.ycombinator.com/.

[2] Reddit. https://www.reddit.com/.

[3] Regular expression library. https://web.archive.org/web/20180920164647/http://regexlib.com/.

[4] Regular expressions 101. https://regex101.com/, 2019.

[5] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017. ISBN 9781509066766. doi: 10.1109/ICSTW.2017.23. URL https://cs.unibg.it/gargantini/research/papers/mutrex_mutation17.pdf.

[6] Alberto Bacchelli, Emma Söderberg, Michal Sipko, Luke Church, and Caitlin Sadowski. Modern code review. 2018. ISBN 9781450356596. doi: 10.1145/3183519.3183525.

[7] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer. page 681, New York, New York, USA, 1 2007. Association for Computing Machinery (ACM). doi: 10.1145/1176617.1176671.

[8] Sebastian Baltes and Stephan Diehl. Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects. *Under submission*, 2018. URL http://arxiv.org/abs/1802.02938.

[9] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Playing regex

golf with genetic programming. pages 1063–1070. Association for Computing Machinery (ACM), 7 2014. doi: 10.1145/2576768.2598333.

[10] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, 5 2016. ISSN 10414347. doi: 10.1109/TKDE.2016.2515587.

[11] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. *EPTCS: Automata and Formal Languages 2014*, 151:109–123, 2014. ISSN 20752180. doi: 10.4204/EPTCS.151.7.

[12] Patrick Biernacki and Dan Waldorf. Snowball Sampling: Problems and Techniques of Chain Referral Sampling. *Sociological Methods & Research*, 10(2):141–163, 11 1981. ISSN 0049-1241. doi: 10.1177/004912418101000205. URL http://journals.sagepub.com/doi/10.1177/004912418101000205.

[13] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work - CSCW '10*, page 301, New York, New York, USA, 2010. ACM Press. ISBN 9781605587950. doi: 10.1145/1718918.1718973. URL http://portal.acm.org/citation.cfm?doid=1718918.1718973.

[14] Raymond P. L. Buse and Thomas Zimmermann. Information Needs for Software Development Analytics. In *Proceedings of the 34th International Conference on Software Engineering*, pages 987–996, Zurich, Switzerland, 2012. IEEE. ISBN 9781467310673. URL https://dl.acm.org/citation.cfm?id=2337343.

[15] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pages 282–293, New York, New York, USA, 2016. ACM Press. ISBN 9781450343909. doi: 10.1145/2931037.2931073. URL http://dl.acm.org/citation.cfm?doid=2931037.2931073.

[16] Carl Chapman and Kathryn T Stolee. Exploring regular expression usage and context in Python. *International Symposium on Software Testing and Analysis (ISSTA)*, 2016. doi: 10.1145/2931037.2931073.

[17] Carl Chapman, Peipei Wang, and Kathryn T Stolee. Exploring Regular Expression Comprehension. In *Automated Software Engineering (ASE)*, 2017.

[18] Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), 2007.

[19] John W Creswell and J David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[20] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[21] James C Davis, Eric R Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *USENIX Security Symposium (USENIX Security)*, 2018.

[22] Mark James Ennis. txt2re. http://www.txt2re.com/, 2006.

[23] Felix Fischer, Konstantin Bottinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? the Impact of Copy&Paste on Android Application Security. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 121–136, 2017. ISBN 9781509055326. doi: 10.1109/SP.2017.31.

[24] Jeffrey EF Friedl. *Mastering regular expressions*. " O'Reilly Media, Inc.", 2006.

[25] Google. Re2 regex engine. https://github.com/google/re2, 2019.

[26] Renáta Hodován, Zoltán Herczeg, and Ákos Kiss. Regular expressions on the web. In *International Symposium on Web Systems Evolution (WSE)*, 2010. ISBN 9781424486366. doi: 10.1109/WSE.2010.5623572.

[27] Barbara A. Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*. 2008. ISBN 9781848000438. doi: 10.1007/978-1-84800-044-5{_}3.

[28] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings - International Conference on Software Engineering*, pages 344–353, 2007. ISBN 0769528287. doi: 10.1109/ICSE.2007.45.

[29] Eric Larson. Automatic Checking of Regular Expressions. In *Source Code Analysis and Manipulation (SCAM)*, 2018.

[30] Eric Larson and Anna Kirk. Generating Evil Test Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2016. ISBN 9781509018260. doi: 10.1109/ICST.2016.29. URL https://pdfs.semanticscholar.org/abbb/a5867872ab723b272a13607b649f6e7bf008.pdf.

[31] Eric Larson and Anna Kirk. Generating Evil Test Strings for Regular Expressions. In *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 309–319. Institute of Electrical and Electronics Engineers Inc., 7 2016. ISBN 9781509018260. doi: 10.1109/ICST.2016.29.

[32] Paul Luo Li, Andrew J. Ko, and Jiamin Zhu. What makes a great software engineer? In *Proceedings - International Conference on Software Engineering*, volume 1, pages 700–710. IEEE Computer Society, 8 2015. ISBN 9781479919345. doi: 10.1109/ICSE.2015.335.

[33] Just Great Software Co. Ltd. Regexmagic. https://www.regexmagic.com/autogenerate.html, 2014.

[34] David R. MacIver. What is property based testing? https://hypothesis.works/articles/what-is-property-based-testing/.

[35] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 12 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837. URL http://ieeexplore.ieee.org/document/1702388/.

[36] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding Relevant Functions and Their Usages. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 111–120. Association for Computing Machinery, 2011. ISBN 1558-1225. doi: 10.1145/1985793.1985809.

[37] Audris Mockus. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS'07*, 2007. ISBN 0769529615. doi: 10.1109/FLOSS.2007.10.

[38] Anders Møller. dk. brics. automaton–finite-state automata and regular expressions for java, 2010, 2010.

[39] Santanu Paul and Atul Prakash. A Framework for Source Code Search Using Program Patterns. Technical Report 6, 1994.

[40] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. pages 102–111. Association for Computing Machinery (ACM), 5 2014. doi: 10.1145/2597073.2597077.

[41] Roger Pressman. Software Engineering: A Practitioner's Approach. chapter Process Models, pages 30–64. McGraw-Hill, seventh edition edition, 2010. ISBN 978–0–07–337597–7.

[42] Asiri Rathnayake and Hayo Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. Technical report, 2014.

[43] S.C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. pages 64–73. Institute of Electrical and Electronics Engineers (IEEE), 11 2002. doi: 10.1109/metric.1997.637166.

[44] Steven P. Reiss. Semantics-based code search. In *Proceedings - International Conference on Software Engineering*, pages 243–253, 2009. ISBN 9781424434527. doi: 10.1109/ICSE.2009.5070525.

[45] Alex Roichman and Adar Weidman. VAC - ReDoS: Regular Expression Denial Of Service. *Open Web Application Security Project (OWASP)*, 2009.

[46] rust lang. Rust regex engine. https://github.com/rust-lang/regex, 2019.

[47] Georgia Robins Sadler, Hau-Chen Lee, Rod Seung-Hwan Lim, and Judith Fullerton. Research Article: Recruitment of hard-to-reach population subgroups via adaptations of the snowball sampling strategy. *Nursing & Health Sciences*, 12(3):369–374, 9 2010. ISSN 14410745. doi: 10.1111/j.1442-2018.2010.00541.x. URL http://doi.wiley.com/10.1111/j.1442-2018.2010.00541.x.

[48] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: a case study. pages 191–201. Association for Computing Machinery (ACM), 8 2015. doi: 10.1145/2786805.2786855.

[49] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Automated Software Engineering (ASE)*, 2018. ISBN 9781450359375.

[50] Henry Spencer. A regular-expression matcher. In *Software solutions in C*, pages 35–71. 1994.

[51] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. pages 20–26. Association for Computing Machinery (ACM), 7 2012. doi: 10.1145/2318202.2318207.

[52] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the Search for Source Code. *ACM Transactions on Software Engineering and Methodology*, 23(3):1–45, 5 2014. ISSN 1049331X. doi: 10.1145/2581377.

[53] Sublime Team. Sublime search and replace. http://docs.sublimetext.info/en/latest/search_and_replace/search_and_replace_overview.html, .

[54] Visual Studio Code Team. Visual studio code - basic editing. https://code.visualstudio.com/docs/editor/codebasics, .

[55] Ken Thompson. Regular Expression Search Algorithm. *Communications of the ACM (CACM)*, 1968.

[56] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. *International Conference on Software Testing, Verification and Validation (ICST)*, 2010. ISSN 2159-4848. doi: 10.1109/ICST.2010.15.

[57] Peipei Wang and Kathryn T Stolee. How well are regular expressions tested in the wild? In *Foundations of Software Engineering (FSE)*, 2018. ISBN 9781450355735.

[58] Peipei Wang, Gina R Bai, and Kathryn T Stolee. Exploring Regular Expression Evolution. Technical report.

[59] Peipei Wang, Gina R Bai, and Kathryn T Stolee. Exploring Regular Expression Evolution. In *Software Analysis, Evolution, and Reengineering (SANER)*, 2019.

[60] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9705, pages 322–334, 2016. ISBN 9783319409450.

[61] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, (May):1–37, 2018. ISSN 15737616. doi: 10.1007/s10664-018-9634-5.

[62] Valentin Wustholz, Oswaldo Olivo, Marijn J H Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2017.

[63] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are Online Code Examples Reliable? An Empirical Study of API Misuse on Stack Overflow. In *International Conference on Software Engineering (ICSE)*, 2018. ISBN 9781450356381. doi: 10.1145/3180155.3180260. URL https://doi.org/10.1145/3180155.3180260.

[64] Inc. Zoom Video Communications. Zoom. https://zoom.us/, 2019.

Appendices

Appendix A

Survey Instrument

Informed Consent

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Informed Consent for Participants in Research Projects Involving Human Subjects

Title of Project: Understanding developer practices for regular expressions

Investigators:

Dongyoon Lee, PhD	dongyoon@vt.edu
Francisco Servant, PhD	fservant@vt.edu
James Davis	davisjam@vt.edu
Louis Michael	louism@vt.edu

Research summary and request for consent

Thank you for your participation in our research study. The purpose of this survey is to better understand the habits of professional software developers as related to regular expressions (regexes). The survey will take around 10 minutes to complete.

To participate, you should have professional software development experience and should have worked with regular expressions before.

This study has been approved through Virginia Tech's WIRB approval process. If you have any questions about your rights as a research subject, you can contact them at 1-800-562-4789 or help@wirb.com. The data collected will be kept confidential and

stored on a secure server. If you have any questions about the study, please contact Dr. Dongyoon Lee (dongyoon@vt.edu), one of the researchers leading the study. By completing this survey, you are providing your consent for your anonymized responses to be used in published research.

Compensation for participation in this survey is provided in the form of a $5 Amazon gift card that is emailed to you by one of the researchers after you complete the survey (24-48 hours). We reserve the right to withhold compensation if responses to the questions appear to be automated.

Thank you for your help!

If you have:
Worked as a **professional software developer**, know what a **regular expression (regex)** is, have used a regular expression in **practice**, and **consent.**
Then please complete the CAPTCHA below and move to the next page.

CAPTCHA

Type the text

Privacy & Terms

Regex Re-use

Page **1** of **4**

This section is about your experience **re-using (i.e., copy-paste or copy-paste and modify)** regular expressions.

Think about your typical practices when putting regular expressions into your code.
What percentage of the time would you estimate that these regular expressions are **re-used from another source (i.e., copy-paste or copy-paste and modify)** such as other source code or Stack Overflow, instead of written yourself "from scratch"?

- ◯ 0% (I never re-use; I always write from scratch)
- ◯ About 25% (I sometimes re-use)
- ◯ About 50% (I re-use about half the time)
- ◯ About 75% (I re-use most of the time)
- ◯ 100% (I always re-use; I never write from scratch)

Why have you in the past **decided to re-use regular expressions** as opposed to writing them yourself from scratch? Please select all that apply

- ☐ ... because I needed a regular expression for a very common purpose
- ☐ ... because I knew of a trusted source where I could probably find a regular expression to re-use
- ☐ ...because I believed that a re-used regular expression would be of higher quality than what I would write
- ☐ ... because I believed that a re-used regular expression would be better tested than my own testing
- ☐ ... because I trusted my abilities to validate the re-used regular expression
- ☐ ... because I had many inputs that I could use to validate the re-used regular expression
- ☐ Other factor #1

  [ ]
- ☐ Other factor #2

  [ ]

- [ ] Other factor #3

- [ ] Other factor #4

- [ ] Other factor #5

In the past why have you selected a **particular** regular expression as the right one to reuse?

- [ ] ... because it was easy to understand.
- [ ] ... because it was well documented; I understood clearly how the regular expression worked from its documentation.
- [ ] ...because it was short.
- [ ] ... because it was designed for my use case, e.g. "a regex for email".
- [ ] ... because it did not need to be modified at all.
- [ ] ...because modifying it for my use case seemed easy.
- [ ] ...because it was from a source that I trust.
- [ ] Other factor #1

- [ ] Other factor #2

- [ ] Other factor #3

- [ ] Other factor #4

☐ Other factor #5

Can you tell us more about your **thought process** when you **decide to re-use a regular expression** vs **writing it from scratch**?

Where do you **re-use** regular expressions from? Please **rank** based on how frequently you use the source, 1 being the most frequent source.

	Most Frequent	2	3	4	5	6 (Least Frequent)	I do not use this source
Stack Overflow	○	○	○	○	○	○	○
RegExLib	○	○	○	○	○	○	○
Other code I have written	○	○	○	○	○	○	○
Other code someone at my company has written	○	○	○	○	○	○	○
Other code, e.g. open-source code	○	○	○	○	○	○	○
Other(s)	○	○	○	○	○	○	○

Is there anything else you would like to say about how you **re-use** regular expressions?

Regex Concerns

Page **2** of **4**

This section is about **concerns and validation** strategies associated with using regular expressions.

When you re-use (copy-paste and possibly modify) a regular expression, estimate the percentage of the time you **know** the regular expression was written **in the same language** in which you will use it.

- ◯ 0% (I never know the source language of the regular expression, or they are never the same language)
- ◯ About 25% (I sometimes know that the source language and destination language are the same)
- ◯ About 50% (Around half of the time I know that the source language and destination language are the same)
- ◯ About 75% (Most of the time I know that the source language and destination language are the same)
- ◯ 100% (I always know that the source language and destination language are the same)

Which of the following have you ***actually* worried about or encountered** in the past when **copy-pasting** a regular expression? (check all that apply)

	Worries I have **worried** about this	Problems I have actually **encountered** this **problem**
Syntactic Differences For example, have you worried that something you copied from stack overflow would not "**compile**" (unsupported syntax/features) in your code?	☐	☐
Semantic Differences For example, have you worried that a regular expression you copied from Stack Overflow would **behave differently** in your code then it was described on Stack Overflow?	☐	☐
Performance Impact Have you ever worried that a regular expression you copy pasted would slow down your code?	☐	☐
Other(s)	☐	☐

For each one of the concerns that you selected above, how did you address them?

How do you **validate** a regular expression, i.e. how do you make sure that it does what you want it to do? (select all that apply)

- [ ] I write unit tests
- [ ] I test with sample input by hand
- [ ] I ask a colleague
- [ ] I use other tools (please list)
- [ ] Other:

Does your validation strategy change when **re-using** a regular expression instead of writing it from scratch?

- ◯ Yes, please explain how:
- ◯ No

In your experience, what makes regular expression validation difficult (e.g., lack of tools to assist in validation)?

Is there anything else that you would like to say about your **concerns and validation** strategies associated with using regular expressions?

Final questions

Page **3** of **4**

Some quick questions...

Are you familiar with the concept of Regular Expression Denial of Service (REDoS)?
(REDOS is also known as "catastrophic backtracking" or "super-linear regular expression behavior")

- ○ Yes
- ○ No

Have you ever modified a regular expression to improve its runtime performance?

- ○ Yes
- ○ No

Have you ever deployed an application that was bottlenecked by regular expression evaluations? (e.g. scientific computing, big data analysis, etc.)

- ○ Yes
- ○ No

Is there a person on your team who is the "regular expression wizard" -- the go-to person for all regular expression questions?

- ○ Yes
- ○ No

Are you the "regular expression wizard" on your team?

- ○ Yes
- ○ No

Do you feel as if code review impacted how you use regular expressions?

- ◯ Yes, please explain how:
- ◯ No
- ◯ My team does not do code review
- ◯ I don't work on a team

When you design a regular expression, is your design influenced by the programming language in which it will be executed?

- ◯ No -- I think that regular expressions are handled the same way in different programming languages.
- ◯ No -- I simply design them using the syntax that I know, even if it is possible that different programming languages may execute them differently
- ◯ No, other:
- ◯ Yes, I normally design regular expressions by consulting the specific syntax of the programming language in which they will be executed
- ◯ Yes, other:

Is there anything else that you would like to add about any of the topics that we have discussed?

Background

Page **4** of **4**

This section covers **background** with regular expressions and some wrap up logistics.

You previously noted that you do not worry about **performance impact** when copy pasting regular expressions. You also stated that you have modified a regular expression to improve its runtime **performance**.

Can you please expand on why you do not worry about **performance** when copy pasting even if you have needed to make **performance improvements** to regular expressions in the past.

You previously noted that you do worry about **semantic differences** when copy pasting regular expression. You also stated that your design of regular expressions **is not** influenced by the programming language in which it will be executed.

Can you please expand on why you worry about **semantic differences** when copy pasting even if you think that regular expressions are handled the same

way in different programming languages.

How long have you worked as a professional software developer?

- ◯ Less than one year
- ◯ 1-2 years
- ◯ 3-5 years
- ◯ 6-10 years
- ◯ more than 10 years

Indicate your experience with programming languages, and with regular expressions in those languages. Leave blank any languages in which you have no experience.

	How long have you programmed in this language?	How long have you used regular expressions in this language?
C/C++		
C#		
Java		
Perl		
PHP		
Python		
Ruby		
JavaScript		

	How long have you programmed in this language?	How long have you used regular expressions in this language?
TypeScript		
Go		
Rust		
Shell e.g. Bash		
Text Editor/IDE		
Other #1		
Other #2		

Where did you learn about regular expressions from? (check all that apply)

- [ ] Online Tutorial or Article
- [ ] Reference Book (e.g. O'Reilly)
- [ ] On the Job Experience Through Co-Workers
- [ ] Technical Training Such as a Coding Bootcamp
- [ ] Undergraduate Course in Computer Science
- [ ] Graduate Level Course in Computer Science
- [ ] Undergraduate Course in Another Field
- [ ] Graduate Level Course in Another Field
- [ ] Other:

Estimate your regular expression expertise level:

- ◯ **Novice**. For example, you know what repetition operators like * and + do.
- ◯ **Intermediate**. For example, you have used more sophisticated features like non-greedy quantifiers */a+?/* and character classes */ \d | \w | [abc] | [^\d]/*.
- ◯ **Expert**. For example, you have used features like backreferences */(a+) \1/* and look-ahead/behind assertions */(?<=abc)def/*.
- ◯ **Master**. You have written a regular expression engine.

Estimate the last time you used regular expressions in a professional context:

- ◯ In the last week
- ◯ In the last month
- ◯ In the last year
- ◯ More than a year ago

What is the size of the company you most recently worked at as a professional developer?

- ◯ Small (100 employees or less)
- ◯ Medium (101 - 999 employees)
- ◯ Large (1000 employees or more)

Is there anything else you would like to say about your **background and general use** of regular expressions such as **how or why you use regular expressions**?

If you are interested in compensation, provide your email. We will only use this to distribute compensation.

If you are interested in reading the results of our study, provide your email. We will only use this to distribute results.

Optional: If you would be willing to participate in a follow-up interview over Zoom (~30 minutes), list the best way to contact you. Interview participants will be compensated an additional $15

Appendix B

Interview Procedure

The first page of questions was covered in all interviews but some respondents were asked additional questions and topics based on time, these are covered in the remainder of the procedure. Some questions are highlighted with a note about when they were added, these questions were added to the procedure after some interviews mentioned these ideas and they were deemed important enough to expound on more constantly.

Main Questions

1. What is challenging about deciding whether to write a regex vs. reusing a regex vs. writing your own non-regex code for a string matching problem?
    a. Other people have mentioned their regex problem being simple as both a reason to write their own regex and as a reason to reuse it instead. Would you attribute this finding to the decision being challenging or to instead different people's preferences?
    b. How do you address/solve these challenges?
2. What is challenging about composing regexes? How do you address/solve these challenges?
    a. Match too much vs. match too little? (added 5/6)
3. What is challenging about reusing regexes?
    a. Other people have mentioned challenges in all these aspects: (1) describing the problem itself that the regex needs to solve, (2) finding a regex to solve that problem, (3) assessing which of the found regexes is the right one to reuse, and (4) dealing with syntax differences between the reused-regex programming language and the programming language in which it will be used. Could you comment in these particular challenges?
    b. How do you address/solve these challenges?
4. What is challenging about validating regexes?
    a. Other people have mentioned challenges in all these aspects: (1) assessing whether enough of the input space has been tested, (2) assessing their runtime/performance. Could you comment in these particular challenges?
    b. How do you address/solve these challenges?
5. What is challenging about maintaining/modifying existing regexes in code?
    a. Other people have mentioned challenges in all these aspects: (1) reading regexes, (2) documenting regexes, and (3) code-reviewing regexes. Could you comment in these particular challenges?
    b. How do you address/solve these challenges?
6. What are your perceptions of regex? added (5/6)

Extra Time Topics/Questions

Background:

- On your survey you indicated you were an expert regarding regular expressions. Can you give me a little bit more background on how you use regex, and how you developed your expertise?

Process:

- What is your typical process when creating a regular expression?
  - Why do you follow this process?
  - Which do you find the most problematic part of this process?
- How is your process different when modifying an existing regex you are not familiar with?
  - What steps do you take to make this process easier or safer?
  - What about the original code helps you in this process? Or make I harder?
- When and how do tools or other resources feature in your process?
  - Why do you use tools in this way/not use tools?
  - Are there any situations where you would be more likely to consider using a tool?
- Could you explain how you usually validate your regular expressions?
  - Do you ever write automated tests for them? Why/why not?
  - How important is it to you to validate regular expressions?
  - Do you ever ask colleagues for input or advice when working with regular expressions?
- How do you obtain sample input for testing regular expressions?
  - Do you preserve any sample input that you use?
  - Have you ever heard of/used input generation tools? (e.g. brics, rex)
- Do you document your regular expressions?
  - Why do you document in this way/not document?
- What is your approach to reviewing code that includes regular expressions?
  - Why do you code review regexes in this way?
  - What are the biggest challenges you face in such a task?
  - What would help you the most? (e.g. documentation, tooling)
- What recommendations would you give an inexperienced developer about regex development?
  - Are there anything's you would advise them against doing?

ReDoS:

- Have you ever had to deal with performance issue in a regular expression?
  - How did/would you go about resolving it?
- Are you familiar with the concept of Regualr Expression Denial-of-Service (ReDoS)?
  - If not, give example
- How would you handle a confirmed report of a ReDoS vulnerability in your code?
  - How serious a risk do you consider ReDoS?

    - What tools or information would help you handle vulnerabilities more effectively?

Regex Reuse:

- What triggers you to want to reuse a regex?
    - Is this similar to what triggers you to want to reuse other code?
    - Once you have started to write a regex what might make you try to reuse one instead?
    - Once you have started trying to find a regex to reuse what might make you want to write one instead?
- Do you encounter any specific issues when trying to reuse a regex?
    - Do you encounter cross language semantic problems?

General Struggles:

- What do you feel is challenging about regex?
    - Are there problems that you have encountered that you solve with tool specifically?
    - Are there issues you felt as a novice that you feel have gone away with your development of more expertise?
    - Are there challenges you still face with regex that you don't have good solutions for?

Appendix C

Interview Consent Form

When participants were contacted to schedule a follow-up interview they were provided with the following consent form which they were asked to review. To confirm that they reviewed the form they were asked to reply to the scheduling email either with a signed copy of the form or with the phrase "I consent to participate in this research".

RESEARCH SUBJECT CONSENT FORM - Interviews

Title: Understanding how and why software developers use, re-use, and struggle with regular expressions

Protocol No.: IRB 18-1110

Sponsor: Virginia Tech

Investigator: Dongyoon Lee, PhD

2228 Knowledge Works II

Blacksburg, Virginia, 24060

USA

Daytime Phone Number: +1 (540) 231-0667

24-hour Phone Number: N/A

You are being invited to take part in a research study. A person who takes part in a research study is called a research subject, or research participant.

What should I know about this research?

- Someone will explain this research to you.
- This form sums up that explanation.
- Taking part in this research is voluntary. Whether you take part is up to you.
- You can choose not to take part. There will be no penalty or loss of benefits to which you are otherwise entitled.
- You can agree to take part and later change your mind. There will be no penalty or loss of benefits to which you are otherwise entitled.
- If you don't understand, ask questions.
- Ask all the questions you want before you decide.

Why is this research being done?

The purpose of this research is to understand how professional software developers use, re-use, and struggle with regular expressions.

About 5-20 subjects will take part in this research.

How long will I be in this research?

We expect that your taking part in this research will last 30-60 minutes.

What happens to me if I agree to take part in this research?

This part of the study is focused on learning from developers' personal experiences when using regular expressions. If you agree to participate in an interview, you will be contacted by a researcher to set up a time that works for you. The interview will be conducted over Zoom, a VoIP software. It will consist of questions about your experiences with regular expressions specifically around your development process. We expect it will take 30-60 minutes. The decision to participate does not affect your relationship with Virginia Tech.

We will share with you any research papers resulting from this project.

What are my responsibilities if I take part in this research?

If you take part in this research, your responsibility will attend an interview at a time and location of your choice that will last 30-60 minutes. We ask that you respond honestly to the interview questions, but you can choose not to answer them or leave the interview at any time.

Could being in this research hurt me?

The conduction of this study does not present any foreseeable risk, discomfort, or inconvenience.

Will it cost me money to take part in this research?

No.

Will being in this research benefit me?

You will be compensated $25 for your time.

We cannot promise any benefits to others from your taking part in this research. However, possible benefits to others include better understanding of how developers use and struggle with regular expressions.

What other choices do I have besides taking part in this research?

This research is not designed to diagnose, treat or prevent any disease.

Your alternative is to not take part in the research.

What happens to the information collected for this research?

Your participation will be anonymous and confidential. At no time will the researchers release identifiable results of the study to anyone other than individuals working on the project without your written consent.

As is standard as part of this research method we will share audio of the interviews with your name removed with a third-party transcription service.

We may publish the results of this research. However, we will keep your name and other identifying information confidential.

We protect your information from disclosure to others to the extent required by law. We cannot promise complete secrecy.

Who can answer my questions about this research?

If you have questions, concerns, or complaints, or think this research has hurt you or made you sick, talk to the research team at the phone number listed above on the first page.

This research is being overseen by an Institutional Review Board ("IRB"). An IRB is a group of people who perform independent review of research studies. You may talk to them at (800) 562-4789, help@wirb.com if:

- You have questions, concerns, or complaints that are not being answered by the research team.
- You are not getting answers from the research team.
- You cannot reach the research team.
- You want to talk to someone else about the research.
- You have questions about your rights as a research subject.

What if I am injured because of taking part in this research?

This research only involves answering questions for 30-60 minutes. Therefore, it is unlikely that you will get injured by taking part in this research.

Can I be removed from this research without my approval?

The person in charge of this research can remove you from this research without your approval. Possible reasons for removal include the intentional attempt of the participants to alter voluntarily the results of the study.

What happens if I agree to be in this research, but I change my mind later?

If you decide to leave this research, contact the research team so that the investigator can remove your material from the study.

Will I be paid for taking part in this research?

For taking part in this research, you may be paid up to a total of $25 at the completion of the study. If you withdraw early from the interview we will pro-rate your payment based on the assumption that a complete interview will take 45 minutes.

Statement of Consent:

Your signature documents your consent to take part in this research.

Signature of adult subject capable of consent	Date
Signature of person obtaining consent	Date