Regexpes are Hard: Decision-making, Difficulties, and Risks in Programming Regular Expressions

Louis G. Michael IV  
Virginia Tech  
louism@vt.edu

James Donohue  
University of Bradford  
j.donohue@bradford.ac.uk

James C. Davis  
Virginia Tech  
davisjam@vt.edu

Dongyoon Lee  
Virginia Tech  
dongyoon@vt.edu

Francisco Servant  
Virginia Tech  
fservant@vt.edu

Abstract—Regular expressions (regexes) are a powerful mechanism for solving string-matching problems. They are supported by all modern programming languages and have been estimated to appear in more than a third of Python and JavaScript projects. Yet existing studies have focused mostly on one aspect of regex programming: readability. We know little about how developers perceive and program with regexes, nor the difficulties that they face.

In this paper, we provide the first study of the regex development cycle, with a focus on (1) How developers make decisions throughout the process, (2) What difficulties they face, and (3) How aware are they about serious risks involved in programming regexes. We took a mixed-methods approach, surveying 279 professional developers from a diversity of backgrounds (including top tech firms) for a high-level perspective and interviewing 17 developers to learn the details about the difficulties they face and the solutions they prefer.

In brief, regexes are hard. Not only are they hard to read, our participants said they are hard to search for, they are hard to validate, and they are hard to document. They are also hard to master: the majority of our studied developers were unaware of critical security risks that can occur when using regexes, and those who knew of the risks weren’t sure how to deal with them. Our findings provide multiple implications for future work, including semantic regex search engines for regex reuse, and improved input generators for regex validation.

Index Terms—regex, developer process, qualitative research

I. INTRODUCTION

Regular expressions (regexes) are a text processing tool baked into all modern programming languages as well as popular tools like text editors [48], [49]. Developers frequently incorporate regexes into their software. Estimates suggest that more than a third of JavaScript and Python projects include a regex [13], [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 [13], regex evolution [51], 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 279 professional developers about their regex practices, and support our findings by interviewing 17 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 developers make when working on regexes: to apply a regex or not; to write or re-use a regex; to identify test input; and whether the regex is correct.

We shed light on the difficulties that developers feel when wrestling 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. And we examine the risks of regex programming, with 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 large-scale surveys of 279 professional software developers from a diversity of companies including several major tech firms, to understand their regex practices. We further our findings by interviewing 17 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.

II. BACKGROUND AND RELATED WORK

A. Regex Programming and Risks

Regexes are a string matching tool, used to identify a generalized subsequence of characters within a string. Generally the process that ends with the inclusion of a regex in code written by a professional developer takes 4 overall steps. First the developer identifies or is tasked with a string matching problem that is assessed for its suitability to be solved using a regex. Next developers will move to compose a regex
evaluating the feasibility of reuse and then either authoring a regex from scratch or reusing with or without modification. Then having arrived at a regex that the developer hopes solves their problem they will validate this regex to confirm. If the regex passes validation developers will document it and integrate it into their project. If the regex fails validation developers will attempt to recompose.

Software developers face two major risks when programming regexes: portability and performance. Many regex dialects have emerged over the years [25], with divergent syntaxes and semantics. Developers therefore face portability problems during regex programming, with the risk that the regex they compose or re-use will be executed in a dialect other than the one they anticipate, with unanticipated behavior (e.g., syntax errors or unexpected match behavior).

Developers also face performance problems leading to security risks due to the polynomial or exponential worst-case time complexity of regex matches in most regex engines [16]. These super-linear regexes can expose applications to Regular expression Denial of Service (ReDoS) vulnerabilities [18], [19], which have been reported on dozens of major websites [47], hundreds of major JavaScript projects [21], and thousands of JavaScript, Python, and Java projects [20], [56]. Any software developers who write client-facing regexes face the risk of regex performance problems and ReDoS security vulnerabilities.

B. Empirical Regex Research

Prior research into regexes has been predominantly quantitative, examining regexes in their role as a software artifact. Researchers have empirically examined on regex reuse [28], regex test coverage [53], regex evolution [52], and regex repair [20]. Others have proposed tools for regex programming, e.g., input generation [5], [33], [39], [44], [50], linters [32], and type checking [46]. Although regexes are interesting artifacts, they also represent hours of developer effort that bear qualitative investigation. Our developer-focused investigation of regex programming is orthogonal to quantitative research that treats regexes as a software artifact. Our qualitative efforts will inform the development of new tools that address the problems developers actually face, maximizing the potential impact of regex tool research.

Only two studies have explored the developer side of regex programming, with an emphasis on composition and comprehension. Chapman and Stolee [14] asked 18 professional developers about how often and in what contexts they use regexes. And in a laboratory setting, Chapman et al. [15] performed a fine-grained study on whether their subjects preferred one regex “synonym” over another (e.g., equivalent patterns that use character classes /[ab]/ or disjunctions /a|b/). Our approach is to cast our net broadly, hearing from hundreds of developers from diverse backgrounds to understand coarse-grained issues surrounding process, difficulties, and risks.

C. Developer Perceptions, Practices and Information Needs

We are not the first to apply qualitative methods to shed light on software engineering practices. Prior work in this vein ranges from general engineering perceptions [34] to specific practices like code review [6]. The standard approach is to survey [6], [12], [23], [27] or interview or observe [6], [23], [24], [30], [34] developers who are exposed to the topic of interest. Some studies have also considered interaction artifacts [6], [17].

Our study is the first investigation of developer regex programming practices in this spirit. We are the first to survey developers on many of these regex topics at all, the first to survey developers on regexes at scale, and the first to conduct regex-focused developer interviews of any kind.

III. 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 dangerous 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: What perceptions do developers have about the value and difficulty of regexes?

RQ2: What influences developer decisions when programming regexes?

RQ3: What do developers find difficult about programming regexes?

RQ4: How do developers handle those difficulties in programming regexes?

RQ5: Are developers aware of portability and security 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. We adapted this framework from the general software engineering methodology of defining requirements, writing, validating, and deploying [40], and introduced a re-use stage based on our intuition that regexes are a kind of “function” complex enough to be re-used 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.

IV. RESEARCH METHOD

Overall method and considerations. We followed a mixed qualitative and quantitative approach [17]. We used a qualitative approach to answer most of our questions, but for RQ1 and RQ5 we also asked some quantitative questions. 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.

Since discovery was our goal, we wanted to maximize the number of professional developers whose perspectives we heard through surveys and interviews. Of course, one major difficulty in qualitative software engineering research is persuading enough (busy, highly-paid) subjects to participate to give weight to findings. We therefore prepared two distinct pairs of surveys and interview protocols, with different emphases roughly on the left and right halves of the process framing Figure 1. This allowed us to reach a diverse population of software developers and to ask a diverse set of questions, while also keeping the survey-taking time to a reasonable 17-minute median and the interview times to roughly 30 minutes.

Survey design. We designed our surveys by first having discussions with professional software developers, and following best practices in survey design [29], [45]. Then, we refined the design of each survey through internal iteration, followed by several pilot administrations each.

Both of our final survey instruments included free response questions about the four stages of regex programming that we set out to study (see §III). In these open questions, we asked developers about their thought process when making the decisions that we described in §III and we asked them what they found difficult in programming with regexes. We also asked them to describe the mechanisms that they follow to handle those difficulties. We used these responses to answer our research questions RQ2, RQ3 and RQ4.

Our surveys also included multiple-choice questions about developers’ perceptions about regexes in general, which we used to answer RQ1. Then, we also included multiple-choice questions asking about developers’ awareness of portability and ReDoS risks, and free response questions asking them how they prevent such risks. We used these responses for RQ5.

Finally, our second survey also included three additional questions about regex reuse, the results of which were included in a different paper, which is currently under blinded review [10]. We did not analyze those questions for this paper.

Survey deployment. We deployed our surveys after obtaining approval from our institution’s ethics board, following a two-pronged strategy to maximize the diversity of respondents.

We deployed our first survey internally in ACME (blinded name), a large international media company. We sought participants through an internal ACME advertising campaign and by asking senior engineering staff to promote the survey.

We deployed the second survey at software companies of various sizes. We used snowball sampling [9], [43], contacting professional developers of our acquaintance who work at tens of different software companies including top Fortune 500 companies and asking them to take the survey and propagate it to their colleagues. To further increase the diversity of responses, we also posted the survey on popular Internet message boards frequented by software developers [1], [2]. We compensated legitimate responses with cookies for the first survey and a $5 Amazon gift card for the second one.

We obtained survey responses from 121 developers for our first survey and 158 developers for our second one. The median respondent of the second survey had more than 6 years of professional experience in the first survey and 3-5 years in the second one.

Interview design and deployment. The final question in our surveys was a request for permission to conduct a follow-up interview. We contacted all survey respondents who were willing to be interviewed, and were able to schedule interviews with 17 of them (around 25%). Following common practice to learn more about a phenomenon [55], our interview protocol was semi-structured [35]. We developed interview guides with general topics and questions to be covered instead of an exact set of questions. We focused our interview guides on the decisions that developers make, the difficulties that they face, and the ways in which they handle them when programming regexes. We also asked for clarification on details of the regex programming process hinted at by our survey results. We compensated interview participants from the second interview population with a $25 Amazon gift card, the first set of interview participants were not compensated.

Data analysis. We analyzed the free response questions in our surveys and the transcriptions of our interviews by using the process of coding [31] (also used in grounded theory [4]). For our second survey one author of the paper read all the responses for a question to identify codes into a code book. Afterwards, the author reread and coded all the responses.
Then a different author of this paper then used the code book to perform their own coding of units. Finally, both sets of codes were used to reach agreement. Due to organizational privacy and confidentiality requirements, the first survey was analyzed by one author coding free responses and then repeating this several weeks later using the codes from the other survey. The results were then compared with the original codes to resolve discrepancies.

V. RQ1: WHAT PERCEPTIONS DO DEVELOPERS HAVE ABOUT THE VALUE AND DIFFICULTY OF REGEXES?

![Chart showing perceptions of regex value and difficulty](chart.png)

We initially wanted to confirm that regexes are a technology that developers value and benefit from, as well as to learn developers’ general perceptions about the difficulty of the technology. We found that developers view regexes as a valuable technology, but one that is also difficult (Figure 2).

To understand whether developers believe that regexes are valuable, our first survey asked participants if they agreed with the statement “Regular expressions are valuable to me in my job”. Respondents were nearly unanimous: 88% of developers agreed that regexes were valuable to their job. Developers also generally agreed that “Regular expressions are an important part of software development knowledge”.

But our participants also described regexes as difficult. 65% of respondents agreed that they “found regular expressions daunting when...learning to code”; most developers did not always feel confident in their usage of regexes; and most developers (70%) disagreed with the statement that “regular expressions are more readable than other code”. These initial findings give strong motivation to pursue regex research in general, and prompted us to investigate our subsequent research questions to characterize the particular regex research in general, and prompted us to investigate our expressions are more readable than other code.

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.

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.”

VI. RQ2: WHAT INFLUENCES DEVELOPER DECISIONS WHEN PROGRAMMING_regexes?

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

A. 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.

1) 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.”

B. 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. re-using a regex. And when they opt to re-use, we asked them how they select re-use 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.

1) Writing Regex vs. Reusing Regex: Our participants often said they would re-use 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 re-use regexes where possible to improve reliability, believing that regexes from a trusted re-use 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 re-use was saving time, but they disagreed about
the more efficient strategy. Some favored re-use (“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.”

2) Which Regex Should I Pick for Reuse?: Developers who opt to re-use 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.”.

3) 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 a 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]”.

C. 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.

1) 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.

2) Is this Reused Regex Correct?: Our participants were split on whether and how to change their validation strategy for re-used 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 re-used regexes cautiously, “I am aware of the security implications of using something from public sources”.

D. Documenting a regex

1) How Much Documentation is Required?: Perhaps related to the differing opinions on the readability of regexes that we identified on 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 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”.

VII. RQ3 & RQ4: WHAT DO DEVELOPERS FIND DIFFICULT ABOUT 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. Table II summarizes our findings. As indicated in Table II, 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.

1We 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 §IX.
## Table II
### Developer-reported Difficulties and Handling Methods in Programming Regexes.

<table>
<thead>
<tr>
<th>Stage</th>
<th>RQ3: Difficulty</th>
<th>RQ4: Handling Mechanism for Difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-cutting</td>
<td>A) Understanding the Problem</td>
<td>Read data, break down problem</td>
</tr>
<tr>
<td></td>
<td>B) Understanding the Regex</td>
<td>Using tool support, breaking down regexes, adding documentation</td>
</tr>
<tr>
<td>Composing</td>
<td>C) Searching for Reuse Candidates</td>
<td>Decomposing the regex, searching for similar code, personal regex library</td>
</tr>
<tr>
<td></td>
<td>D) Unfamiliar Syntax</td>
<td>Using tool support reading the regex documentation</td>
</tr>
<tr>
<td>Validating a Regex</td>
<td>E) Testing Edge Cases</td>
<td>Test all available input</td>
</tr>
<tr>
<td></td>
<td>F) Testing Enough Inputs</td>
<td>Work to find or generate more real cases, property testing</td>
</tr>
</tbody>
</table>

### A. Understanding the Problem

1) **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 §VII-C.

2) **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.”

### B. Understanding the Regex

1) **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.

2) **Handling:** Developers reported two ways 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.”

### C. Searching for Reuse Candidates

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

1) **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 participant 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 (§VII-A) and the difficulty of articulating it to a existing search engines.

2) **Handling:** Developers reported three mechanisms 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.”

Anther 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 re-use candidates. This handling mechanism seems similar to having a personal library of utility functions that you copy into your codebase as needed.

D. Unfamiliar Syntax

1) 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.”

2) Handling: Developers relied on two handling mechanisms 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.”

E. Testing Edge Cases

1) Difficulty: Developers also found it difficult to identify corner cases or edge cases, terms colloquially used to refer to boundary values [42]. 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”.

2) 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 therefore 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.”, and went on to say “[but] unfortunately the input that I get can’t be universal!”.

F. Testing Enough Inputs

1) 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”

2) 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 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 generation using property based tests [37].

VIII. RQ5: Are Developers Aware of Portability and Security (ReDoS) Risks When Programming Regexes?

As discussed in §II 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’ awareness of these risks, as well as their handling mechanisms.

A. Portability Risks

1) Awareness: We asked developers whether they worry about a series of risks involved in regex reuse: syntactic, semantic, and performance differences when re-using regexes. We report their answers in Figure 3.

![Fig. 3. Developers responding to the second survey when asked about which they worried about when reusing expressed a range of concerns, emphasizing semantic portability issue, 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 is discussed in §VIII-B.]

![Fig. 3](image-url)
Developers who responded to the second survey worry about syntactic and semantic differences (over 50%), and among 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 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. In fact, some developers reported that they prefer to use regexes over other alternatives because of their (perceived) portability across languages. One survey respondent stated “It is consistent across languages”, and another said that the “same regex can be used across technologies/systems”. In concurrent work we have explored this misconception [10].

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 correctness issues due to improper regex validation. We note that these assumptions can affect correctness whether or not regexes are being re-used, simply as a result of an incorrect mental model for regex behavior.

2) 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: “Transitoning 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 re-use 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.”

B. ReDoS

1) Awareness: Regex open the somewhat obscure security concern of ReDoS. ReDoS is fairly avoidable however. If

\footnote{For example, consider the participants we described across several earlier sections, who re-use from a “trusted source” like Stack Overflow and do not validate carefully as a result.}

proper steps are taken to sanitize input and to not use superlinear regex 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 38% of all developers surveyed knew of the possible vulnerability. This is concerning since the vulnerability is easy to introduce without noticing and slip through validation without seeing an issue. We note that ReDoS is a vulnerability rather than an exploit, and is only relevant if the regex may match against malicious input. 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.

2) 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.

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 [41], [44], [54], [56].

IX. DISCUSSION AND IMPLICATIONS

Our findings have implications for many research directions.

A. 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.

1) 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 [38]. 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.

B. 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 using and the interesting practice of keeping regex lists for reuse. These findings open multiple opportunities for research.

1) 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 §VII-A).

Thus, 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 example 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.

2) 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 [14] proposed a 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.

3) 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.

4) 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.

C. 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.

1) Live Support for Regex Composition: Now that we better understand some of the characteristics that developers value in regexes (see §IX-A), 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.

2) 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 [14] already proposed the idea of regex refactoring. Our findings throw more light into what kinds of refactorings would be desirable for developers.

3) 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 [7], [8], [22], [36] 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.

D. 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.

1) Regex Input Generation for Humans: Many tools have been proposed to generate input for testing regular expressions [5], [33], [39], [14], [50]. 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.

2) 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.
E. 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.

1) 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 Figure 3) 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.

F. Understanding Regexes

Many developers mentioned the terseness and cryptic-ness of the syntax of regexes, and how that makes it very difficult for them to understand them.

1) 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.

X. Threats to Validity

Construct Validity: In order to pose questions about the regex process to developers we first outlined a general process based on our understanding of general software engineering practice. This may have limited the extent to which participants reflected on their own distinct decision making factors or difficulties.

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 reproducibility of this study since other individuals may have interpreted the same information in different ways. This is a very natural limitation of qualitative research and we have strived to limit its impact by corroborating and discussing our findings across authors and across the two populations.

External Validity: We sampled hundreds of developers across two different populations, but this is still a small portion of all software engineers and populations. In the second survey for part of the sample we used snowball sampling starting with professional contacts of the authors, biasing towards people known by the authors and at companies where the authors have worked. Though this may compound some bias we believe our snowball sampling allowed us to contact both more and a much wider range of developers. The information they provided similar to the participants from our public recruitment approaches.

XI. Conclusion

Regexes are powerful tools that developers find valuable. Regexes are also hard to work with. We identify 6 difficulties developers face when using regexes and handling mechanisms they employ to deal with these difficulties. The way that developers cope leaves a great deal of room for improvement. Developers are also critically unaware of risks they take when using regexes with under 40% 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.

Our survey instruments and interview protocols can be found at https://tinyurl.com/RegexAreHardASE2019.

REFERENCES


[29] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. In Qualitative communication research


