Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions

James C. Davis
davisjam@vt.edu
Virginia Tech, USA

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

Christy A. Coghlan*
ccogs@vt.edu
Virginia Tech, USA

Francisco Servant
fservant@vt.edu
Virginia Tech, USA

Dongyoon Lee
dongyoon@vt.edu
Virginia Tech, USA

ABSTRACT

This paper explores the extent to which regular expressions (regexes) are portable across programming languages. Many languages offer similar regex syntaxes, and it would be natural to assume that regexes can be ported across language boundaries. But can regexes be copy/pasted across language boundaries while retaining their semantic and performance characteristics?

In our survey of 158 professional software developers, most indicated that they re-use regexes across language boundaries and about half reported that they believe regexes are a universal language. We experimentally evaluated the riskiness of this practice using a novel regex corpus — 537,806 regexes from 193,524 projects written in JavaScript, Java, PHP, Python, Ruby, Go, Perl, and Rust. Using our polyglot regex corpus, we explored the hitherto-unstudied **regex portability problems**: logic errors due to semantic differences, and security vulnerabilities due to performance differences.

We report that developers' belief in a regex *lingua franca* is understandable but unfounded. Though most regexes compile across language boundaries, 15% exhibit semantic differences across languages and 10% exhibit performance differences across languages. We explained these differences using regex documentation, and further illuminate our findings by investigating regex engine implementations. Along the way we found bugs in the regex engines of JavaScript-V8, Python, Ruby, and Rust, and potential semantic and performance regex bugs in thousands of modules.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**; • **Social and professional topics** → *Software selection and adaptation.*

KEYWORDS

Regular expressions, developer perceptions, re-use, portability, empirical software engineering, mining software repositories, ReDoS

*Christy A. Coghlan is now employed by Google, Inc.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338909

ACM Reference Format:
James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338909

1 INTRODUCTION

Regular expressions (regexes) are a core component of modern programming languages. Regexes are commonly used for text processing and input sanitization [105], appearing, for example, in an estimated 30-40% of open-source Python and JavaScript projects [20, 26]. However, crafting a correct regex is difficult [101], and developers may prefer to re-use an existing regex than write it from scratch. They might turn to regex repositories like RegExLib [4, 7]; or to Stack Overflow, where "regex" is a popular tag [8]; or to other software projects. For example, a regex derived from the Node.js `path` module appears in more than 2,000 JavaScript projects [26].

Correctness and security are fundamental problems in software engineering in general, and for regexes in particular: *re-using regexes can be risky.* Like other code snippets [109], regexes may flow into software from Internet forums or other software projects. Unlike most code snippets, however, regexes can flow unchanged across language boundaries. Programming languages have similar regex syntaxes, so re-used regexes may compile without modification. However, surface-level syntactic compatibility can mask more subtle *semantic* and *performance* portability problems. If regex semantics vary, then a regex will match different sets of strings across programming languages, resulting in logical errors. If regex performance varies, a regex may have differing worst-case behavior, exposing service providers to security vulnerabilities [25, 78].

Despite the widespread use of regexes in practice, the research literature is nearly silent about regex re-use and portability. We know only anecdotally that some developers struggle with "[regex] inconsistencies across [languages]" [20]. In this paper we explored the coupled concepts of cross-language regex re-use and regex portability using a mixed-methods approach. First, we surveyed 158 professional developers to better understand their regex practices (§4), and empirically corroborated the regex re-use practices they reported (§6). Then, we investigated the extent to which these practices may result in bugs. We empirically measured semantic and performance portability problems, attempted to explain these problems using existing regex documentation, and explored regex

engine implementations to illuminate our findings (§7). We are not the first to observe regex portability issues, but we are the first to provide evidence of the extent and impact of the phenomenon.

Our contributions are:

- We describe the regex re-use practices of 158 developers (§4).
- We present a first-of-its-kind *polyglot regex corpus* consisting of 537,806 unique regexes extracted from 193,524 software projects written in 8 popular programming languages (§5).
- We empirically show that regex re-use is widespread, both within and across languages and from Internet sources (§6).
- We identify and explain semantic and performance regex portability problems (§7). We report that approximately 15% of the regexes in our corpus have semantic portability problems, while 10% have performance portability problems. Most of these problems could not be explained using existing regex documentation.
- We identify thousands of potential regex bugs in real software, as well as bugs in JavaScript-V8, Python, Ruby, and Rust (§8).

2 BACKGROUND

2.1 Regex Dialects

Most programming languages support regexes, providing developers with a concise means of describing a set of strings. There has been no successful specification of regex syntax and semantics; Perl-Compatible Regular Expressions (PCRE) [46] and POSIX Regular Expressions [47] have influenced but not standardized the various regex dialects that programming languages support, leading to manuals with phrases like "these aspects...may not be fully portable" [6]. Anecdotally, inconsistent behavior has been reported even between different implementations of the same regex specification [19].

This lack of standardization may come as no surprise to developers familiar with regexes as a library feature rather than a language primitive. But for the latest generation of developers, regexes have always been part of the programming language, and because the regex dialects are similar in syntax it would be natural for developers to assume that they are in fact a *lingua franca*. We report that many developers do make this assumption, and we explore the potential consequences by investigating the distinct semantic and performance characteristics of many regex dialects.

2.2 Regex Denial of Service (ReDoS)

Under the hood, programming languages implement a *regex engine* to test candidate *inputs* for membership in the language of a regex. Most regex engines evaluate a regex match by simulating the behavior of an equivalent Finite Automaton (Deterministic or Non-) on the candidate string [90], but they vary widely in the particular algorithm used. Despite the recommendations of automata theorists [24, 95], in most programming languages a regex match may require greater-than-linear time in the length of the regex and the input string. Such a super-linear (SL) match may require polynomial or exponential time in the worst case [25, 78].

SL regex behavior had long been considered an unlikely attack vector in practice, but in the past year this has begun to change. Davis *et al.* [27] and Staicu and Pradel [92] identified Regular expression Denial of Service (ReDoS) as a major problem facing Node.js applications, and Davis *et al.* reported thousands of SL regexes affecting over 10,000 JavaScript projects [26]. Although it is known that SL regex behavior is possible in JavaScript, Python, and Java [26, 104, 107], from a portability perspective we do not know the relative risk of ReDoS across different programming languages. Cox [24] has suggested that languages fall into two classes of performance, though he did not systematically support his claim.

2.3 Developer Practices Around Regexes

Despite the widespread use of regexes in practice [20, 26], surprisingly little is known about how software developers write and maintain them. Recent studies have shed some light on the typical languages developers encode in regexes [20, 26], the relative readability of different regex notations [21], developer regex test practices [102], and developer regex maintenance practices [26, 101].

Most of these works have focused on software artifacts rather than on developers' thought processes and day-to-day practices. The only previous qualitative perspective on developers' approach to regex development is Chapman and Stolee's exploratory survey of 18 professional software developers employed by a single company [20]. They reported high-level regex practices like the frequency with which those developers use regexes and the tasks they use regexes for.

3 RESEARCH QUESTIONS

In this work we seek to better understand *developer regex re-use practices* and understand the *potential risks* they face. First, we survey professional software developers to learn their regex perceptions and practices. We then measure regex re-use practices in real software to corroborate the findings of our survey. Finally, we empirically evaluate the semantic and performance portability problems that may result from cross-language regex re-use practices, and explain differences across languages. Our research questions:

Theme 1: Developer perspectives

RQ1: Do developers re-use regexes?

RQ2: Where do developers re-use regexes from?

RQ3: Do developers believe regexes are a *lingua franca*?

Theme 2: Measuring regex re-use

RQ4: How commonly are regexes re-used from other software?

RQ5: How commonly are regexes re-used from Internet sources?

Theme 3: Empirical portability

RQ6: *Semantic portability:* When and why do regexes match different sets of strings in different programming languages?

RQ7: *Performance portability:* When and why do regexes have different worst-case performance in different programming languages?

4 THEME 1: DEVELOPER PERSPECTIVES

We surveyed developers to better understand regex re-use and portability issues from their perspective.

> **Findings:** (RQ1) 94% of developers re-use regexes,
> (RQ2) commonly from Stack Overflow and other code.
> (RQ3) 47% of developers treat regexes like a *lingua franca*.

4.1 Methodology

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 re-use

practices; and (3) what awareness they have of regex portability problems[1]. We drafted our survey based on discussions with professional software developers, and followed best practices in survey design [51, 88]. We refined the survey through internal iteration and two pilot rounds with graduate students.

Survey deployment. After obtaining approval from our institution's ethics board, we took a two-pronged approach to surveying professional developers. First, following the snowball sampling methodology [16, 81], we asked developers of our acquaintance to take the survey and propagate it to their colleagues. Second, to diversify our population, we posted the survey on popular Internet message boards frequented by software developers (HackerNews [1] and Reddit [3] (`r/SampleSize`, `r/coding`, and `r/compsci`). We compensated respondents with a $5 Amazon gift card.

Filtering results. We received some invalid responses from users on the Internet message boards. We manually inspected the first 100 responses to develop filters for validity. We report on responses that took at least 5 minutes, were internally consistent, and gave a "thoughtful" answer to at least one of our open-ended questions. This filtered out 253 responses, mostly from a spoofing campaign.

4.2 Results

Demographics. We received 158 valid 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[2] (Figure 1).

RQ1: Re-use Prevalence. Almost all (94%) of respondents indicated that they re-use regexes, with 50% indicating that they *re-use* a regex at least half of the time that they *use* a regex (Figure 2 (a)). The most frequent reason to re-use a regex was to meet a common use case, *e.g.*, matching emails. This supports a previous hypothesis that developers may write regexes for a few common reasons [26]. Participants also mentioned time savings: "A good programmer doesn't re-invent the wheel."

RQ2: Re-use Sources. Developers most frequently said they re-use regexes from Stack Overflow, but they often re-use regexes from other code, including their own, a colleague's, or open-source projects (Figure 2 (b)). About 90% of respondents reported re-using regexes from some Internet source, and about 90% reported re-using regexes from other code.

RQ3: Developer Perception of *Lingua Franca*. We asked developers if their regex design process was influenced by language. Figure 3 (a) shows that 47% of our respondents do not have a design process that is language specific. And their actions match their beliefs: as shown in Figure 3 (b), respondents frequently re-use regexes without being confident that they were written in the same language. Only 21% of respondents (34/158) were confident they never re-used across language boundaries.

[1]Due to space limits we do not report all results.

[2]Regex skill was self-reported on a scale from novice to master, based on familiarity with increasingly complex regex features according to Friedl [40]. "Intermediate: For example, you have used more sophisticated features like non-greedy quantifiers (`/a+?/`) and character classes (`/\d|\w|[abc]|[^\d]/`)."

5 POLYGLOT REGEX CORPUS

In order to answer our remaining research questions we needed a *polyglot regex corpus*: a set of regexes extracted from a large sample of software projects written in many programming languages. The existing regex corpuses are small-scale [20, 107] or include only two programming languages [26]. Our corpus is neither, covering about 200,000 projects in 8 programming languages — see Table 1.

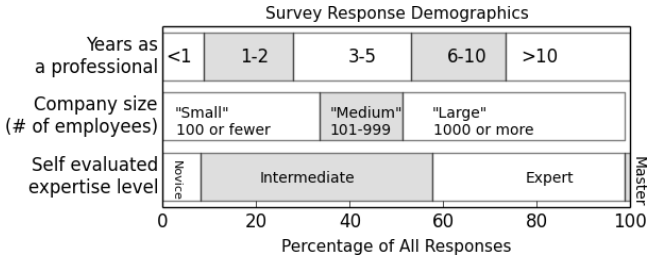

Figure 1: Our survey reached a diverse set of developers.

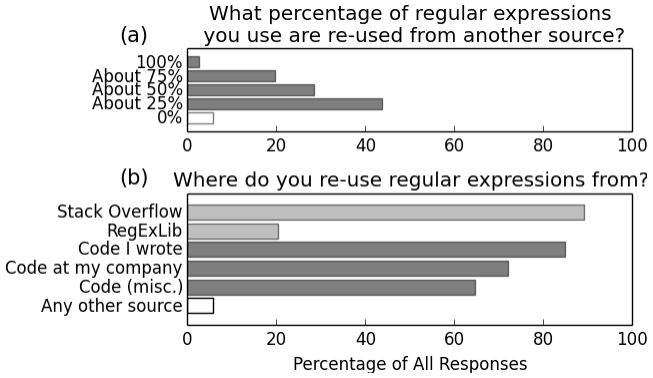

Figure 2: (a) When developers must use a regex, they frequently re-use them from another source. (b) Developers commonly re-use from the Internet and other software.

Languages. We are interested in studying common regex practices, and as a result we focus our attention on "major" programming languages defined by two conditions: (1) The language has a large module ecosystem; (2) The language is widely used by the open-source community. We operationalized these concepts by consulting the ModuleCounts website [28] and the GitHub language popularity report [42]. We also considered Go, Perl, and Rust for scientific interest; Perl popularized the idea of regexes as a first-class language member, and Go and Rust are relatively new mainstream languages.

Software projects. Within these languages, we chose to study the software modules published in each language's primary *module registry* for two reasons. First, it permits a relatively fair cross-language comparison, since we observe that many modules fill equivalent ecological niches, *e.g.*, logging or schema validation. Second, we feel that modules are of greater general interest than applications. Modules are published, maintained, and used by a mix of open-source and commercial software developers, and bugs and security vulnerabilities in modules have a significant ripple effect.

[3]We also extracted regexes from TypeScript source code, by transpiling it to JavaScript.

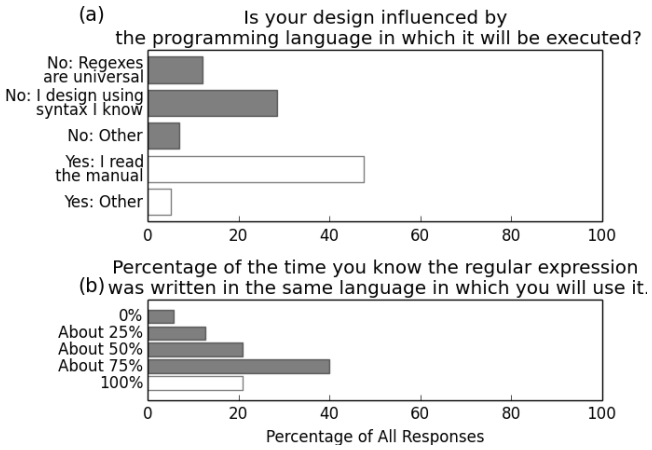

Figure 3: (a) Many developers design regexes without considering the programming language. (b) Developers' regex re-use decisions also imply belief in regex as a *lingua franca*.

Table 1: Our regex corpus was derived from software written in 8 programming languages. The first five languages are ranked by the most available libraries (ModuleCounts [28]) and popularity in open-source (GitHub). We also studied Go, Perl, and Rust out of scientific interest. The two final columns show the contribution to our corpus.

Lang. (Registry)	Libs.	GH	# mod. anal.	Unique regexes (avg.)
JavaScript[3] (npm)	1	1	24,997	150,922 (6.0)
Java (Maven)	2	3	24,986	19,332 (0.8)
PHP (Packagist)	3	5	24,995	44,237 (1.2)
Python (pypi)	4	2	24,997	43,896 (1.8)
Ruby (RubyGems)	5	4	24,999	153,334 (6.1)
Go (Gopm)	9	9	24,997	22,105 (0.9)
Perl (CPAN)	7	—	31,827 (all)	142,777 (4.5)
Rust (Crates.io)	10	—	11,724 (all)	2,025 (0.2)
		Sum:	*193,524*	*578,628*

Our goal was to analyze the most important modules in each language's primary module registry. To have a uniform measure of importance across languages, we filtered each registry for the modules available on GitHub, sorted those by the number of stars, and analyzed the top 25,000 modules from each registry. Borges and Valente recently showed that GitHub star count is a reasonable proxy for importance [17]. Unsurprisingly, we found that the distribution of GitHub stars was similar for the modules in each language, and analyzing the top 25,000 modules typically captured all but the (very long) tail of 0-2 stars. Perl and Rust had relatively few modules in their registries, and we analyzed all of their modules.

Regex extraction. Following [26], for each module we cloned the HEAD of its default branch from GitHub and extracted any statically-declared regexes. We extracted regexes declared in regex evaluations as well as regexes compiled and stored in variables for later use. In each module we extracted regexes only from source files in the language corresponding to the registry, omitting regexes in places like build scripts written in another language.

Polyglot regex corpus. Our corpus contains 537,806 unique regexes extracted from 193,524 projects written in 8 programming languages. Each language's contributions are listed in Table 1. Average regex use varies widely by language, from 0.2 regexes per module (Rust) up to 6.1 regexes per module (Ruby). The total unique regexes by language exceeds 537,806 due to inter-language duplicates (§6).

6 THEME 2: MEASURING REGEX RE-USE

The developers in our survey indicated that they re-use regexes from other software and from Internet sources like Stack Overflow. They also reported re-using regexes across language boundaries. In this theme we corroborate their report by measuring the extent of regex re-use — modules that use non-unique regexes.

Definition of re-use. To the best of our knowledge we are the first to attempt to measure regex re-use. As a first approximation, in keeping with the phrasing in our survey ("copy/pasting regexes"), we label as re-use *any pair of identical regexes* (string equality). To eliminate trivially identical regexes like `/\s/`, we conservatively require any matching regexes to be at least 15 characters long. While it is possible that two developers might independently produce the same (longer) regex, this seems unlikely given that hundreds of distinct regexes have been reported even for "simple" languages like emails [26]. We do not consider less strict measures of regex equivalence like string [101] or behavioral [20] similarity, though such measures might better capture the "Ship of Theseus" approach to regex re-use described by some of our survey respondents.

> **Findings:** (RQ4) Thousands of corpus modules (20%) share the same complex regexes, both within and across languages. (RQ5) 5% of all corpus modules (about 10,000), primarily in JavaScript, use regexes from Stack Overflow and RegExLib.

6.1 RQ4: Re-use from Other Software

How much intra-/inter-language regex re-use occurs in our corpus?

6.1.1 Methodology. When developing our regex corpus (§5), we tracked the modules and registries in which each regex was found. As noted above, we only consider as re-use candidates the regexes that are at least 15 characters long. When such a regex appeared in multiple modules in the *same* registry, we mark those modules as containing an intra-language duplicate. When such a regex appeared in at least one module in *different* registries, we mark those modules as containing an inter-language duplicate. Note, then, that for a single duplicated regex we may mark several modules as containing intra-language duplicates and/or inter-language duplicates.

6.1.2 Results. The extent of intra- and inter-language regex re-use by modules is shown in Figure 4 (second and third bars). Developers re-use regexes in the modules in every language, some more than others. In most languages, 10% or more of the modules contain an intra-language duplicate, and inter-language duplicates are also common. These duplicates are often due to "popular" regexes. For example, we found the 16-character "<email>:" regex `/[\w\-]+\@([^:]+):/` in 476 modules.

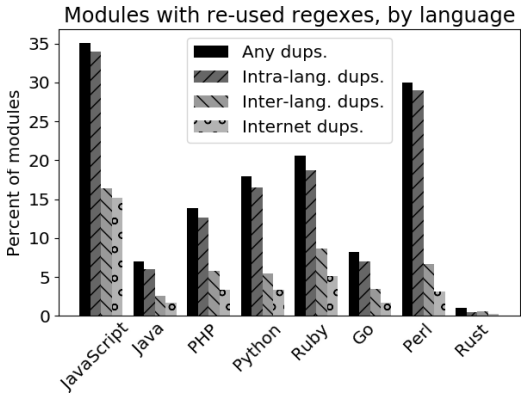

Figure 4: Empirical regex re-use practices, by language.

6.2 RQ5: Re-use from Internet Sources

The developers in our survey frequently indicated that they re-use regexes from one of two Internet sources: RegExLib [4] and Stack Overflow [5]. Next we use our corpus to corroborate their claims.

6.2.1 Methodology. We extracted regexes from RegExLib and Stack Overflow, and then searched our corpus for matches. In both of these relatively-unstructured Internet regex sources, the resulting set of "regexes" may include false positives; it is the intersection of our corpus and these sets that is of interest. An intersection is a case where a real regex from our corpus also appeared verbatim in one of these Internet sources. Any module that contained one of these (15 characters or longer) Internet regexes was marked as containing an Internet duplicate.

RegExLib regexes. We obtained a reasonably complete dump of the RegExLib database by searching for "all regexes".

Stack Overflow regexes. For Stack Overflow, we relied on the "regex" tag to identify regexes. Through manual analysis we found that questions and posts with the "regex" tag commonly denote regexes using code snippets. Using all Stack Overflow posts as of September 2018[4], we found all questions tagged with "regex" as well as the answers to those questions and automatically extracted code snippets from those posts. To filter, we then removed snippets that contained no regex-like characters based on Table 4 of [20].

6.2.2 Results. Our findings are shown in Figure 4 (fourth bar for each language). Many of the modules in our corpus contain at least one Internet regex. This practice is most common in JavaScript — 15% of npm modules contain an Internet regex.

7 THEME 3: EMPIRICAL PORTABILITY

Having shown that developers re-use regexes across language boundaries, now we experiment on our polyglot regex corpus to investigate the implications of copy/pasting a regex from one language to another. First we consider semantic portability (§7.1), then performance portability (§7.2).

Experimental parameters. These experiments were performed on a 10-node cluster of server-class nodes. We used the same base tools in both experiments: a tester for each of the 8 languages. Each tester accepts a regex pattern and input and attempts a partial regex match. Table 2 lists the language versions used in our tests.

When we compare a regex's behavior in a pair of languages, we use the subset of the regex corpus that is syntactically valid in that pair. This simulates the regex re-use practices we identified. Most comparisons are on the majority of the corpus — 76% of the corpus was valid in every language, and 88% were valid in all but Rust.

Table 2: Summary of language versions and docs used in our experiments. Most are at the default for Ubuntu 16.04.

Language	Version information	Documentation
JavaScript	Node.js v10.9.0 (V8 v6.8)	[31, 32]
Java	Oracle JDK 8	[23]
PHP	PHP 7.4.0-dev (cli)	[44]
Python	Python 3.5.2	[37, 53]
Ruby	Ruby 2.3.1p112	[18]
Go	Go v1.6.2	[43]
Perl	Perl v5.22.1	[2, 54, 97]
Rust	Rust v1.32.0 (nightly)	[30]

Findings: (RQ6) 15% of regexes exhibit documented and undocumented *semantic* differences. (RQ7) 10% of regexes exhibit *performance* differences due to regex engine algorithms and optimizations.

7.1 RQ6: Semantic Portability Problems

When two languages express the same feature using different syntax, developers face a translation problem. But when two languages exhibit different features (or behaviors) for the same syntax, developers must solve a semantic problem. In this section we empirically study the semantic portability problems that developers may face.

7.1.1 Methodology. To understand variations in regex semantics, we tested the behavior of each regex in our corpus on a variety of inputs in each language of interest. Any inconsistent regex behavior across languages is something a developer would have to discover and address after re-using the regex.

Input generation. In search of an interesting set of inputs, we created an ensemble of five state-of-the-art regex input generators: Rex [100], MutRex [11], EGRET [56], ReScue [87] and Brics [65]. These generators produce either matching strings (Rex, Brics) or both matching and mismatching strings (MutRex, EGRET, ReScue). We used Rex, MutRex, and EGRET unchanged. We modified ReScue to use the strings it explores in its search for SL inputs. We modified Brics to generate random input subsets, not infinitely many inputs.

We wanted these inputs to provide good regex automaton coverage. Wang and Stolee showed that 100 Rex-generated inputs yield about 50% regex coverage [102], so we requested 10,000 inputs from each input generator with a time limit of 10 seconds. Table 3 summarizes the number of unique inputs generated for each regex.

Attempted match. For each regex, for each input, for each programming language that supported it, we tested for a match using partial-match semantics[5]. On a match, we recorded (1) the substring that matched, and (2) the contents of capture groups.

[4]See https://archive.org/download/stackexchange/stackoverflow.com-Posts.7z.

[5]We used default flags. As the 8 languages in our study support around 20 distinct regex flags, evaluating a meaningful subset of the flag combinations was infeasible.

Table 3: Statistics for semantic portability experiment.

Metric	Value
Percentile inputs per regex: 25^{th}-50^{th}-75^{th}	1,057-2,410-2,510
Regexes with any difference witnesses	15.4% (82,582)
Regexes with any match witnesses	8.1% (43,417)
Regexes with any substring witnesses	4.2% (22,597)
Regexes with any capture witnesses	7.5% (40,457)

Witnesses. Some pairs of languages may perfectly agree on the behavior of a regex on all of its inputs; others may not. We refer to (*regex*, *input*) pairs that produce different behavior in different programming languages as *difference witnesses* between those languages, and distinguish between three disjoint types of witnesses:

(1) *Match witness*: Languages disagree on whether there is a match.
(2) *Substring witness*: Languages agree that there is a match but disagree about the matching substring.
(3) *Capture witness*: Languages agree on the match and the matching substring, but disagree about the division of the substring into any capture groups of the regex.

7.1.2 Results. Table 3 summarizes our results. About 15% of regexes participated in at least one difference witness, and among the language pairs we observed all three classes of witnesses. In Table 3 and Figure 5 we report the number of distinct regexes participating in the difference witnesses rather than the number of distinct witnesses themselves, because we expect that many of the witnessing inputs for a given regex are members of an equivalence class on which a difference manifests.

A more detailed description of the semantic differences between languages is presented in Figure 5. The cells are colored proportional to the number of regexes that have any witness of a difference between that pair of languages. The three numbers in the cell denote the percent[6] of regexes with match, substring, and capture witnesses for that pair of languages. As can be seen in Figure 5: there are many language pairs with match witnesses; PHP and Python are the primary sources of substring witnesses; and PHP is the primary source of capture witnesses.

7.1.3 Analysis. We developed an automatic tool, the `Cross Examiner`, to estimate the causes of the difference witnesses identified through our experiment. We iteratively examined unclassified witnesses, referenced the regex documentation for the disagreeing languages (Table 2), understood the reason for the different behaviors where documented, and encoded heuristics to classify witnesses as due to this behavior. The causes we identified are summarized in Table 4. Approximately 98% (80,736/82,582) of witnesses could be explained by one or more of these causes.

Table 4 differentiates the witnesses by type. The first group of witnesses are cases where some languages support a feature that others do not. In the second group, languages use the same syntax for different features. The third group are cases where languages use the same syntax for the same features but exhibit different behavior. The final group are bugs we identified, described below.

[6] At the scale of our corpus, each percentage point represents about 5,300 regexes.

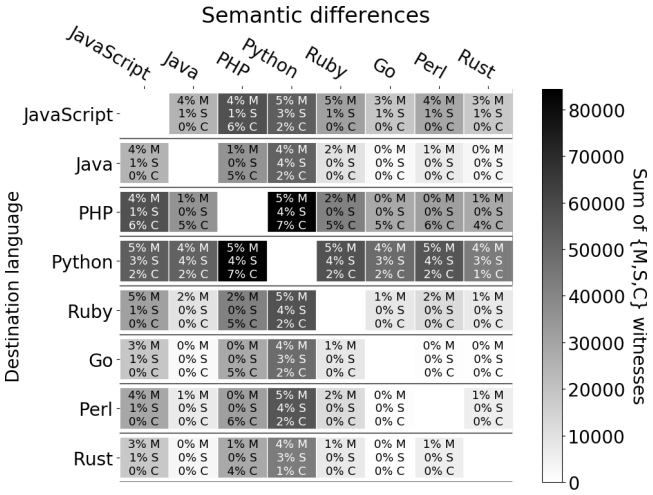

Figure 5: (Symmetric) Pairwise view of difference witnesses by language and type. The individual cells indicate the percent of the regex corpus with at least one (M)atch, (S)ubstring, and (C)apture witnesses in that language pair, and darker cells indicate that regexes more commonly have difference witnesses in that pair of languages. For example, Java, Go, and Rust generally agree on regex behavior.

Documented semantics. We studied each language's regex documentation (Table 2) to see if these witnesses could be easily explained. Comparing the grey cells and boldfacing in Table 4, we note that more than half of the "unusual" behaviors were unspecified in that language's documentation. **Testing, not reading the manual, is the only way for developers to learn these behaviors.**

7.1.4 Regex Engine Testing. Though in this experiment we assumed that the regex engines were trustworthy, our methodology can be viewed as a mix of fuzz [22] and differential [62] testing. Under a *lingua franca* hypothesis, if languages disagree then at least one of them is wrong. During our examination of difference witnesses, we identified five cases where one language disagreed with the others *and* its behavior was inconsistent with the corresponding regex documentation. We opened bug reports based on the behaviors briefly described in the third section of Table 4. So far the bugs have been confirmed in V8-JavaScript, Python, Ruby, and Rust.

7.2 RQ7: Performance Portability Problems

Programming languages have distinct regex engines which may exhibit different performance characteristics. A re-used regex might have worse worst-case performance in its new language than in its language of origin. For example, software being ported from PHP to Node.js might develop Regular expression Denial of Service (ReDoS) vulnerabilities because regexes often have worse worst-case performance in Node.js. In this experiment, we measure the frequency with which regexes have different worst-case performance characteristics in different programming languages.

7.2.1 Methodology. We generally follow the methodology of Davis *et al.* [26] and use their tools In brief, for each regex we (1) query an ensemble of state-of-the-art super-linear regex detectors, and

Table 4: Difference witnesses identified during our semantic portability experiment. Each row indicates a witness regex, the expected behavior(s), and each language's interpretation. The first three groups describe different classes of valid but semantically distinct behavior. The final group describes the bugs we found; E- means Engine, D- means Docs. Boldface indicates potentially-surprising behavior (cf. §8). "-" indicates languages where a feature causes syntax errors. The behavior in the grey cells was not specified in the documentation.

Witness	Description	JavaScript	Java	PHP	Python	Ruby	Go	Perl	Rust
False friends 1: Regex notation describes a feature in one language and no feature in another.									
`/\Qa\E/`	Quote directive ; "QaE"	**"QaE"**	Quote	Quote	**"QaE"**	**"QaE"**	Quote	Quote	–
`/\G/`	Match assertion ; "G"	**"G"**	Assertion	Assertion	**"G"**	Assertion	–	Assertion	–
`/\Ab\Z/`	Anchors ; "AbZ"	**"AbZ"**	Anchors	Anchors	Anchors	Anchors	–	Anchors	–
`/a\z/`	End of line ; "az"	**"az"**	EOL	EOL	**"az"**	EOL	EOL	EOL	EOL
`/\K/`	Match reset ; "K"	**"K"**	–	Reset	**"K"**	Reset	–	Reset	–
`/\e/`	ESC ; "e"	**"e"**	ESC	ESC	**"e"**	ESC	–	ESC	–
`/\cC/`	ctrl-C ; "cC"	ctrl-C	ctrl-C	ctrl-C	**"cC"**	ctrl-C	–	ctrl-C	–
`/\x{41}/`	"A" (hex) ; "x...x"	**"x...x"**	"A"	"A"	–	–	"A"	"A"	"A"
`/(a)\g1/`	Backref notation ; "ag1"	**"ag1"**	–	Backref	**"ag1"**	**"ag1"**	–	Backref	–
`/(a)\g<1>/`	Backref notation ; "ag<1>"	**"ag<1>"**	–	Backref	**"ag<1>"**	Backref	–	–	–
`/\p{N}/`	Unicode digit ; "pN"	**"p{N}"**	1	1	**"p{N}"**	1	1	1	1
`/\pN/`	Unicode digit ; "pN"	**"pN"**	Digit	Digit	**"pN"**	**"pN"**	Digit	Digit	Digit
`/[[:digit:]]/`	Digit ; Custom Char. Class (CCC)	**CCC**	**CCC**	Digit	**CCC**	Digit	Digit	Digit	Digit
False friends 2: The same regex notation describes different features.									
`/^a/`	^: Beginning of input or line	Input	Input	Input	Input	**Line**	Input	Input	Input
`/a++/`	Possessive quantifier ; regular	–	Possessive	Possessive	–	Possessive	–	Possessive	**Regular**
`/(a)\1/`	Backref ; octal	Backref	Backref	Backref	Backref	Backref	–	Backref	**Octal**
`/\h/`	Horz. whitespace; Hex; "h"	**"h"**	Whitespace	Whitespace	**"h"**	**Hex**	–	Whitespace	–
Nuanced: The same regex notation describes the same feature, but engines exhibit subtly different behavior.									
`/(a)(?<b>b)/`	Named and unnamed capture groups?	Both	Both	Both	–	**Named only**	–	Both	–
`/[]]/`	CCC of "]" ; empty CCC + "]"	**Empty**	"]"	"]"	"]"	"]"	"]"	"]"	"]"
`/((a*)+)/`	Diff. capture of \2 on "aa"	**\2: "aa"**	\2: empty	\2: empty	\2: empty	\2: empty	**\2: "aa"**	\2: empty	**\2: "aa"**
`/((a)|(b))+/`	Diff. capture of \2 on "ab"	**Empty**	"a"	"a"	"a"	"a"	"a"	"a"	"a"
Bugs we found in regex engines.									
E-Python: `/(ab|a)*?b/`	Diff. capture of \1 on input: "ab "	"a"	"a"	"a"	**Empty**	"a"	"a"	"a"	"a"
E-Rust: `/(aa$)?/`	Matched substring on "aaz"	Empty	Empty	Empty	Empty	Empty	Empty	Empty	**"aa"**
E-Rust: `/(a)\d*\.?\d+\b/`	Matched substring on "a0.0c "	"a0"	"a0"	"a0"	"a0"	"a0"	"a0"	"a0"	**"a0.0"**
E-JavaScript: *Complicated*	Input order matters?	**Yes**	No	No	No	No	No	No	No
D-OracleJava: `/$\s+/`	$ matches before final \r?	No	**Yes**	No	No	No	No	No	No
D-Ruby: `/a{2}?/`	Lazy "aa" ; optional "aa"	Lazy	Lazy	Lazy	Lazy	**Optional**	Lazy	Lazy	Lazy

then (2) evaluate any predicted super-linear regex behaviors in each language of interest using partial-match semantics.

Experimental parameters. We allowed each of the detectors to evaluate a regex for up to 60 seconds using no more than 2 GB of memory. If a detector predicted that a regex would be super-linear, we evaluated its proposed worst-case input in each of the 8 languages in our study using input strings intended to trigger exponential or polynomial behavior[7]. If a regex match took more than 10 seconds in some language, we marked it as super-linear.

Reducing false positives. We extended their methodology in two ways to reduce the number of false negatives (*i.e.,* SL regexes marked as linear-time). First, we added Shen *et al.*'s new dynamic SL regex detector [87] to their ensemble ([75, 104, 107]). More critically, we introduce a new technique that identifies both polynomial and exponential SL regexes that their detector ensemble would not detect. The static detectors in their ensemble: (1) assume full-match semantics, and (2) do not scale well to regexes with large NFAs. We combat these problems by querying detectors with the original regex as well as *regex variants* that they can more readily analyze.

The first query variant addresses an *unrealistic assumption* in the analysis performed by some of the detectors in the ensemble ([75, 104, 107]). Although these detectors assume that the regex engine is using full-match semantics, regex engines generally default to partial-match semantics. For example, some detectors predict linear behavior for `/a+$/`, but it is quadratic in many languages when used with a partial-match API. To address this assumption, we query the detector ensemble with an (anchored) full-match variant of unanchored regexes, *e.g.,* `/^[\s\S]*?a+$/`.

The second query variant addresses *inefficient implementations* in the detector ensemble. Some of the detectors exceed our time limit on regexes with large NFA representations. For example, they time out on the (exponential) regex `/(a{1,1000}){1,1000}$/` because its NFA explodes in size. To account for this inefficiency, we query the detector ensemble with variants that replace bounded quantifiers with unbounded ones, *e.g.,* `/(a+)+$/`.

These variants reduce the rate of false negatives without introducing false positives. Although we query the detector ensemble on several variants, we always test any worst-case input on the original regex (dynamic validation). The first variant may unmask polynomial regexes that would otherwise go undetected, and the second may identify both polynomial and exponential regexes.

7.2.2 Results. Figure 6 illustrates the extent to which the regexes in our polyglot corpus exhibit worst-case super-linear behavior in each of the 8 languages under study.

Figure 6 indicates that SL regexes may be more common — by up to an order of magnitude! — than was previously reported [26]. The

[7] We used 100 pumps for exponential and 100,000 pumps for polynomial.

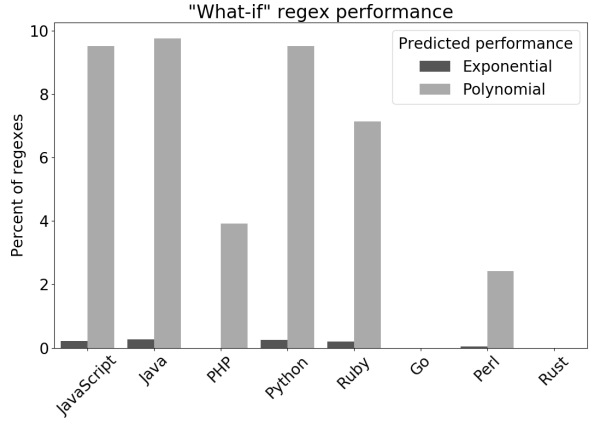

Figure 6: Proportion of SL regexes in each language. There are three distinct families of worst-case regex performance. We identified no regexes with exponential behavior in Go and Rust, and only 6 regexes had polynomial behavior in those languages. Regexes with exponential behavior are rare in PHP and Perl (Perl – 227; PHP – 0), but polynomial behavior still occurs. In contrast, over 1,000 regexes have exponential behavior in Ruby, Java, JavaScript, and Python, and polynomial behavior is also more common in those languages.

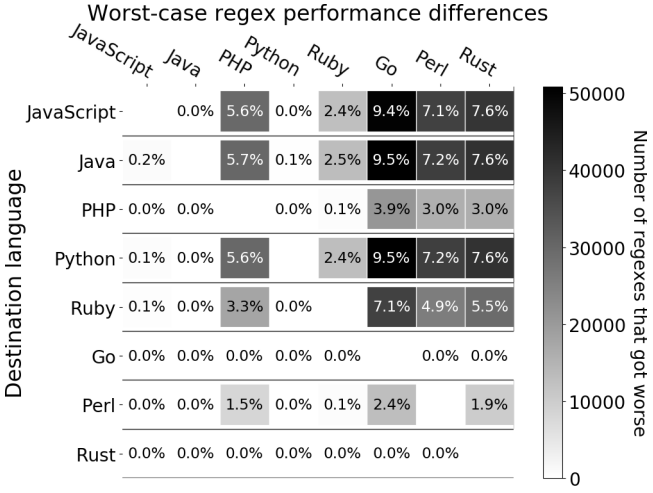

Figure 7: Pairwise view of regex performance differences. Cells are colored according to the number of regexes that exhibit worse behavior in the destination (row) than the hypothetical source (column). Darker rows are dangerous destinations; the individual cells contain the percent of the regexes supported in that language pair whose worst-case performance is worse in the destination. For example, regexes do not perform any worse in JavaScript than Java, but 8% of regexes perform worse when moved from Rust to JavaScript.

majority of the newly-discovered regexes were identified through our variant testing technique; as expected, the new detector by Shen *et al.* [87] identified only exponential regexes (1,421 of them). Our results agree with a small-scale estimate in Java [107]. Although Figure 6 does not provide a direct comparison to [26], the same larger proportions occur when considering the subset of our corpus derived from JavaScript and Python (as theirs was).

7.2.3 Analysis. The proportion of regexes that exhibit exponential and polynomial worst-case behavior varies widely by language. The regex engines in these languages appear to fall into three families: (1) *Slow* (JavaScript, Java, Python, Ruby); (2) *Medium* (PHP, Perl); and (3) *Fast* (Go, Rust). To clarify this taxonomy, Figure 7 shows the frequency with which regexes exhibit *worse* behavior in one of a pair of languages. For example, we see that the ~10% of regexes that are super-linear in both Java and JavaScript (cf. Figure 6) are the *same* regexes. The worst-case performance of a regex generally worsens when moved between these families, but not within them.

In this section we explore the reasons behind these three families of regex performance. We studied the language documentation and the implementation of these engines and identified a variety of mechanisms by which some regex engines fall prey to super-linear behavior and others avoid it.

Documented performance. We studied each language's regex documentation (Table 2) to see whether its worst-case performance is discussed. JavaScript, Java, and Python only provide tips on minor optimizations. PHP and Ruby comment vaguely on worst-case performance: "can take a long time to run" [44]. The best documentation explicitly states worst-case expectations: linear (Rust and Go) or exponential (Perl). Similar to the semantic behaviors in Table 4, in most languages these performance differences can only be identified through experimentation.

Under the hood. The primary distinction between these families is their core regex matching algorithms and varying support for super-linear regex features (*e.g.*, backreferences [9]). Go and Rust offer linear behavior because they primarily rely on Thompson's algorithm for linear-time regex evaluations [95], though in consequence they offer a limited set of regex features. In contrast, the remaining 6 languages perform regex matches using some variant of Spencer's backtracking algorithm [91]. Thompson's algorithm is similar to a breadth-first traversal of the NFA graph, while Spencer's is analogous to a depth-first traversal. Some implementations of the Spencer-style DFS may exhibit super-linear behavior due to redundant state visits, though there are also truly exponential (NP-complete) regexes with backreferences [9, 15].

Within the set of Spencer engines, though, there are distinct Medium and Slow Families. In our experiments, exponential behavior was unusual in PHP and Perl, while it occurs at about the same rates in Java, JavaScript, Python, and Ruby. Similarly, PHP and Perl have a lower incidence of polynomial behavior than do the other Spencer engines. The differences between these two families can be attributed to a mix of *defenses* and *optimizations*.

To the best of our knowledge, PHP and Perl are the only Spencer engines in our study that have explicit defenses against exponential-time behavior. Both languages rely on *counters* to track the amount of work performed during a match, and if a regex evaluation exceeds a threshold it is terminated with an exception. In experiments, we found that these counters are incremented such that exponential searches may trigger the threshold but poly-time searches will not. Perl additionally maintains a *cache* of visited states in order to short-circuit redundant paths through the NFA, permitting it to evaluate

some searches in linear time that take polynomial or exponential time in other Spencer engines.

In addition to their exponential defenses, PHP and Perl both have optimizations that act as a safeguard against polynomial regex engine behavior. For partial matches, some regex engines will try every possible starting offset in the string, trivially leading to polynomial behavior. PHP and Perl have optimizations to prune these starting offsets, and these optimizations appear to reduce the incidence of polynomial behavior in those languages. The relevant optimizations seem to be: (1) skipping ahead to plausible starting points, and (2) filtering out inputs that lack necessary substrings.

To the best of our knowledge, this is the first description of these real-world regex engine mechanisms in the scientific literature[8]. We hope our findings will inform the maintenance and development of regex engines that are less susceptible to super-linear behavior.

Three families, not two? In our experiments we were surprised to find three families of regex engine performance instead of the two previously described by Cox [24]. Perhaps based on Cox's analysis, others argued that exponential regex behavior in Java would translate to PHP [87]. The defenses and optimizations we identified in PHP and Perl have previously gone unremarked.

8 REGEX BUGS

Semantic bugs. Although developers may identify some semantic regex problems during testing, others may cause unexpected regex behavior in practice. To estimate the frequency of semantic problems in practice, we developed linter-style tools to identify regexes that use features that are unavailable in their language (Table 4). For example, in JavaScript the anchor notation `/\Ab\Z/` is interpreted literally as AbZ, but developers who use this notation in JavaScript projects probably intend anchors. Among the JavaScript (npm) modules from which we derived our corpus, we identified 31 modules that used this notation. In total we identified hundreds of modules containing potential semantic regex bugs. We have begun opening bug reports against these modules.

It is possible that these regexes were derived from copy/paste practices. However, developers might introduce such bugs even when designing regexes from scratch, since they may design them based on a (supposed) regex *lingua franca* that does not extend to the language in which they are developing (cf. Figure 3).

ReDoS regexes. The super-linear regexes we identified represent potential ReDoS vectors. After filtering out regexes that appear in paths like `test` or `build`, we have initiated the responsible disclosure process to inform the developers of 14,495 modules about potential security vulnerabilities.

9 DISCUSSION AND FUTURE WORK

Considerations: software engineers. Our findings suggest that porting regexes across language boundaries, *e.g.,* from other software projects or from Stack Overflow, is a potentially risky activity. Subtle semantic and performance issues can occur and should be considered by developers introducing regexes into their code. Unfortunately, the largest developer communities are in the languages most vulnerable to ReDoS (cf. Table 1 and Figure 6).

We have released our many-language tools to help developers understand the possible risks of regexes. Our tools can test the semantic and performance of regexes in many languages on many inputs. We hope Table 4 will be a useful reference for developers.

Recommendations: programming language designers. We empathize with the developers we surveyed who expected regexes to behave consistently across programming languages. We believe that regexes should truly be the *lingua franca* many developers already believe them to be. We suggest that having the fastest or most feature-rich engine is not worth the cost of regex portability problems. Perhaps supported by researchers, programming language designers could agree on a universal regex specification and relieve software engineers of the burden of reconciling regexes across languages. We acknowledge that diversity and competition sometimes improve outcomes for users, but regexes are a mature technology and unifying their behavior makes sense.

Each language's regex documentation currently focuses only on its own syntax and semantics. We recommend that regex documentation additionally describe its deviations from external specification(s), *e.g.,* PCRE [46] or PX-BRE [47]. Explicitly discussing incompatibilities will inform developers of "gotchas", and it will have the indirect effect of reminding them that regexes are (currently) not a *lingua franca.* Longer term, explicitly considering each language's divergence from specification(s) will help designers reach agreement on a next-generation universal regex specification. Lastly, languages should document their worst-case regex performance.

We recommend that language designers in the "Slow Family" (JavaScript, Java, Python, Ruby) of regex engines adopt techniques from the "Medium Family" (PHP, Perl) to reduce the incidence of ReDoS vulnerabilities in these popular languages.

Corpus applications. Our polyglot regex corpus is a promising basis for further research. Clearly developers search for regexes — can we adapt semantic code search techniques [50] to the discovery of relevant regexes? And do developers have different regex needs in different programming languages — do these differences manifest in measurable ways, and should this affect the regex feature support or optimizations used in regex engines?

Regex tools and regex engines. Motivated by this work, we envision a regex "universal translator" to help developers port regexes between languages. This task is complicated by incomplete regex specifications, different feature support in different programming languages, and performance variations. As a starting point, van der Merwe *et al.*'s work on regex transformations that preserve semantics but change performance seems promising [98].

We believe that two directions for regex engine research are promising. First, we accidentally identified bugs in four regex engines (§7.1.3). Testing regex engines by refining regex semantics and applying model-checking techniques will improve the developer experience. Second, we suggest that most ReDoS vulnerabilities can be solved at the regex engine level by refining and enhancing the optimizations already present in Perl and PHP (§7.2.3). Care will be needed, however, to avoid changing regex semantics.

[8] Besides our description, we are only aware of descriptions of these defenses in discussion forum posts [69] and the source code itself (*e.g.,* see line 7835 of [70]). These mechanisms are not described in the PHP and Perl documentation that we studied.

Other *Lingua Francas*. What is the impact of other "*lingua franca* problems" in software engineering? For example, how do developers account for variations in SQL dialects, Markdown specifications, software compilers, and browser JavaScript support, and what are the consequences when they fail to do so?

10 THREATS TO VALIDITY

Internal validity. *Survey.* Our survey instrument has not been validated [51]. We assume the survey respondents who survived our "bogus response" filter replied in good faith.

Performance portability. Our results assume that the SL regex detector ensemble is effective. These detectors were designed with the naive Spencer-style regex engines in mind ("Slow family") and might miss SL regexes in the Medium and Fast families. For example, it is not clear whether the defenses of PHP and Perl are sound or simply effective against these detectors' inputs.

External validity. *Regex corpus.* Our methodology for procuring the regex corpus faces two threats. First, our corpus is composed only of statically-declared regexes. To generalize, we assume that either most regexes are statically declared, or that dynamically-declared regexes have similar properties. Second, we only extract regexes from modules. We do not know whether developers follow the same regex practices when writing regexes in modules and in applications, so our results may not generalize to applications.

Construct validity. *Regex re-use.* We took a simple approach to identifying regex re-use in our corpus: exact string matches for regexes at least 15 characters long. We chose this threshold based on our assessment of regexes more or less likely to have been independently derived by multiple developers. However, there may have been shorter re-used regexes, longer independently-derived regexes, and many regexes that were re-used with modifications to tailor them to specific use cases.

Our definition of re-use does not account for the possibility of wholesale file duplication, which is not true regex re-use. File duplication would only affect our intra-language regex re-use results.

11 RELATED WORK

Empirical studies of regexes. The empirical study of regex use is a recent endeavor, with several lines of research. Chapman and Stolee assessed the use of different regex features in Python [20]. Using that corpus, Chapman *et al.* assessed the relative understandability of regex synonyms to determine community preferences [21]. Wang and Stolee reported that regexes are *poorly unit tested* in Java applications [102], though this might be due to developer processes not captured by version control, *e.g.,* using regex checking tools [55]. Our polyglot regex techniques will enable generalizing some of these results to other programming languages.

Software re-use: other code. Software re-use is a prevalent practice in software engineering [14, 33, 39, 86]. Developers re-use code from their own or other projects [89], introducing *code clones* [41, 79]. Multiple studies estimate that more than 50% of the code in GitHub is duplicated [58, 64], with similar ratios for Android applications [80].

Whether code clones are good practice is a matter of debate. Some researchers have pointed out the benefits of code clones [68], and found little difference between cloned and non-cloned code in qualities such as comprehensibility [83] and defect-proneness [74, 84]. But other studies have examined negative effects of cloning code [49], such as maintenance difficulties [38] due to frequent [59] but inconsistent [52] changes. As a result, a wide variety of techniques have been proposed to detect code clones, *e.g.,* [76, 79, 82].

To the best of our knowledge, our paper presents the first study of regex re-use and the problems that can arise from it.

Software re-use: Internet forums. Researchers have also studied software re-use from Internet forums. Multiple studies found evidence of code flow from Stack Overflow to software repositories [109], and found that code frequently flows, although sometimes without respecting license terms [10] or authorship attribution [12]. Researchers have also studied the interplay of developer contributions to both resources [99].

Given the prevalence of code snippets in Stack Overflow, multiple tools have been proposed to help developers re-use them, *e.g.,* to automatically generate comments [106], or to augment the IDE [72, 73]. However, some problems have been identified with reusing code snippets from Stack Overflow, *e.g.,* quality [67] and usability [108]. Furthermore, other studies have identified particular threats with code re-use from Stack Overflow, such as API misuse [110], security vulnerabilities [36, 63], or unreadable code [96].

In this paper we found that Internet forums are also a popular source of regex re-use among developers, and we observed similar risks: feature mis-use and ReDoS vulnerabilities.

Migration. Researchers have long discussed the difficulties of code migration [29, 48, 60, 77, 94, 103]. As new technologies emerge, so do new migration tools, *e.g.,* within [13] and between languages [34, 61, 66, 71, 85, 93, 111] and frameworks [35, 45, 57].

Our work shows that regexes are (currently) not a *lingua franca*, creating an opportunity for tools for regex migration.

12 CONCLUSION

Regexes are not a *lingua franca*. Although about 92% of regexes will compile in most programming languages, their apparent portability masks problems of correctness and performance. We empirically investigated the extent and causes of these portability problems, offering the first empirical perspective on regex portability. In the process we identified hundreds of modules with potential semantic problems and thousands with potential performance problems, plus documentation and implementation errors in popular languages.

Unfortunately, but quite understandably, about half of the software developers we surveyed believe and act as though regexes are a *lingua franca*. We hope that this paper increases developer awareness of regex portability problems. We also hope to motivate language designers toward regex standardization.

REPRODUCIBILITY

An artifact containing our survey instrument, regex corpus, and analyses is available at https://doi.org/10.5281/zenodo.3257777.

ACKNOWLEDGMENTS

A. Kazerouni and R. Davis advised us on data analysis. J. Donohue and E. Deram shared insights about developer re-use practices. This work was supported in part by the National Science Foundation grant CNS-1814430.

REFERENCES

[1] [n.d.]. Hacker News. https://news.ycombinator.com/.
[2] [n.d.]. Perl Regular Expressions - Perl. https://perldoc.perl.org/5.22.0/perlre.html.
[3] [n.d.]. Reddit. https://www.reddit.com/.
[4] [n.d.]. Regular Expression Library. https://web.archive.org/web/20180920164647/http://regexlib.com/.
[5] [n.d.]. Stack Overflow - Regex tag. https://stackoverflow.com/questions/tagged/regex.
[6] 2009. regex(7) - Linux manual page - POSIX.2 regular expressions. http://man7.org/linux/man-pages/man7/regex.7.html.
[7] 2013. Debuggex: A Composable Regex Repository. https://web.archive.org/web/20170222084629/https://www.debuggex.com/blog/2013/a-composable-regex-repository/.
[8] 2018. Tags – Stack Overflow. https://web.archive.org/web/20180919183037/https://stackoverflow.com/tags?tab=popular.
[9] Alfred V Aho. 1990. *Algorithms for finding patterns in strings.* Elsevier, Chapter 5, 255–300.
[10] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. 2017. Stack Overflow: A code laundering platform?. In *International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE.
[11] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2017. MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW).*
[12] Sebastian Baltes and Stephan Diehl. 2018. Usage and attribution of Stack Overflow code snippets in GitHub projects. *Empirical Software Engineering* (2018), 1–37.
[13] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *International Symposium on Software Testing and Analysis (ISSTA).*
[14] Veronika Bauer et al. 2016. Comparing reuse practices in two large software-producing companies. *Journal of Systems and Software* 117 (2016), 545–582.
[15] Martin Berglund and Brink Van Der Merwe. 2017. Regular Expressions with Backreferences. In *Prague Stringology.* 30–41.
[16] Patrick Biernacki and Dan Waldorf. 1981. Snowball Sampling: Problems and Techniques of Chain Referral Sampling. *Sociological Methods & Research* 10, 2 (11 1981), 141–163.
[17] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016
[18] James Britt and Neurogami Secret Laboratory. [n.d.]. Regexp - Ruby. https://ruby-doc.org/core-2.3.1/Regexp.html.
[19] Cezar Câmpeanu and Nicolae Santean. 2009. On the intersection of regex languages with regular languages. *Theoretical Computer Science* 410, 24-25 (2009), 2336–2344.
[20] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. *International Symposium on Software Testing and Analysis (ISSTA)* (2016).
[21] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring Regular Expression Comprehension. In *Automated Software Engineering (ASE).*
[22] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. *Computers & Security* 75 (2018), 118–137.
[23] Oracle Corp. [n.d.]. Pattern - Java. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html.
[24] Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).
[25] Scott Crosby. 2003. Denial of service through regular expressions. *USENIX Security work in progress report* (2003).
[26] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*
[27] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. 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).*
[28] Erik DeBill. [n.d.]. Module Counts. http://modulecounts-production.herokuapp.com/.
[29] Arie van Deursen, Paul Klint, and Chris Verhoef. 1999. Research Issues in the Renovation of Legacy Systems. *Fundamental Approaches to Software Engineering* 1577 (1999), 1–21.
[30] The Rust Project Developers. [n.d.]. regex - Rust. https://docs.rs/regex/1.1.0/regex/.
[31] MDN Web Docs. [n.d.]. RegExp - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp.
[32] MDN Web Docs. [n.d.]. Regular Expressions - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions.
[33] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An exploratory study of cloning in industrial software product lines. In *European Conference on Software Maintenance and Reengineering.* IEEE.
[34] M. El-Ramly, R. Eltayeb, and H.A. Alla. 2006. An Experiment in Automatic Conversion of Legacy Java Programs to C#. *IEEE International Conference on Computer Systems and Applications, 2006.* March (2006), 1037–1045.
[35] Xiaochao Fan and Kenny Wong. 2016. Migrating user interfaces in native mobile applications. In *International Workshop on Mobile Software Engineering and Systems (MOBILESoft).*
[36] Felix Fischer, Konstantin Bottinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? the Impact of Copy&Paste on Android Application Security. In *IEEE Symposium on Security and Privacy (IEEE S&P).* 121–136.
[37] Python Software Foundation. [n.d.]. re – Regular expression operations - Python. https://docs.python.org/3.6/library/re.html.
[38] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.
[39] W. B. Frakes and Kyo Kang. 2005. Software reuse research: status and future. *IEEE Transactions on Software Engineering* 31, 7 (July 2005), 529–536.
[40] Jeffrey EF Friedl. 2006. *Mastering regular expressions.* " O'Reilly Media, Inc.".
[41] Mohammad Gharehyazie, Baishakhi Ray, Mehdi Keshani, Masoumeh Soleimani Zavosht, Abbas Heydarnoori, and Vladimir Filkov. 2018. Cross-project code clones in GitHub. *Empirical Software Engineering* (2018), 1–36.
[42] GitHub. 2018. The State of the Octoverse. https://octoverse.github.com/.
[43] Google. [n.d.]. regexp - Go. https://golang.org/pkg/regexp/.
[44] The PHP Group. [n.d.]. Regexp - PHP. http://php.net/manual/en/regexp.introduction.php.
[45] Ahmed E. Hassan and Richard C. Holt. 2005. A lightweight approach for migrating web frameworks. *Information and Software Technology* 47, 8 (2005), 521–532.
[46] Hazel, Philip. 2018. PCRE - Perl Compatible Regular Expressions. https://web.archive.org/web/20180919101106/https://www.pcre.org/.
[47] IEEE and The Open Group. 2018. The open group base specifications issue 7, 2018 edition, ieee std 1003.1-2017.
[48] Ivar Jacobson and Fredrik Lindström. 1991. Reengineering of old systems to an object-oriented architecture. *ACM SIGPLAN Notices* 26, 11 (1991), 340–350.
[49] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *International Conference on Software Engineering (ICSE).* IEEE.
[50] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2016. Repairing programs with semantic code search. In *Automated Software Engineering (ASE).* 295–306.
[51] Barbara A. Kitchenham and Shari L. Pfleeger. 2008. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering.*
[52] Jens Krinke. 2007. A study of consistent and inconsistent changes to code clones. In *Working conference on reverse engineering (WCRE).* IEEE.
[53] A.M. Kuchling. [n.d.]. Regular Expression HOWTO - Python. https://docs.python.org/3.6/howto/regex.html.
[54] Mark Kvale. [n.d.]. Perl Regular Expressions Tutorial - Perl. https://perldoc.perl.org/5.22.0/perlretut.html.
[55] Eric Larson. 2018. Automatic Checking of Regular Expressions. In *Source Code Analysis and Manipulation (SCAM).*
[56] Eric Larson and Anna Kirk. 2016. Generating Evil Test Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation (ICST).*
[57] Terry Lau, Jianguo Lu, Erik Hedges, and Emily Xing. 2001. Migrating E-commerce Database Applications to an Enterprise Java Environment. In *Conference of the Centre for Advanced Studies on Collaborative Research.*
[58] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages (OOPSLA).*
[59] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. 2007. Evaluating the harmfulness of cloning: A change based experiment. In *Mining Software Repositories (MSR).* IEEE.
[60] Andrew J Malton. 2001. The Software Migration Barbell. *Proceedings of the ASERC Workshop on Software Architecture* (2001).
[61] J. Martin and H.a. Muller. 2002. C to Java migration experiences. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering* (2002), 143–153.
[62] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[63] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. 2018. Secure coding practices in java: Challenges and vulnerabilities. In *International Conference on Software Engineering (ICSE).* IEEE.

[64] Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS'07*.

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

[66] M. Mossienko. 2003. Automated Cobol to Java recycling. In *Conference on Software Maintenance and Reengineering (CSMR)*, Vol. 7. IEEE, 40–50.

[67] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *IEEE International Conference on Software Maintenance (ICSM)*. IEEE.

[68] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File cloning in open source java projects: The good, the bad, and the ugly. In *IEEE International Conference on Software Maintenance (ICSM)*. IEEE.

[69] PerlMonks. [n.d.]. Perl regexp matching is slow?? https://perlmonks.org/?node_id=597262.

[70] PerlMonks. [n.d.]. Snapshot of Perl 5 regex.c. https://web.archive.org/web/20190206210240/https://github.com/Perl/perl5/blob/blead/regexec.c.

[71] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. 2017. Statistical migration of API usages. In *International Conference on Software Engineering Companion (ICSE-C 2017*.

[72] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack overflow in the ide. In *International Conference on Software Engineering (ICSE)*. IEEE Press.

[73] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Working Conference on Mining Software Repositories (MSR)*. ACM.

[74] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. 2012. Clones: What is that smell? *Empirical Software Engineering* 17, 4-5 (2012), 503–530.

[75] Asiri Rathnayake and Hayo Thielecke. 2014. *Static Analysis for Regular Expression Exponential Runtime via Substructural Logics*. Technical Report.

[76] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[77] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *Automated Software Engineering (ASE)*. IEEE.

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

[79] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.

[80] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. 2012. Understanding reuse in the android market. In *IEEE International Conference on Program Comprehension (ICPC)*. IEEE.

[81] Georgia Robins Sadler, Hau-Chen Lee, Rod Seung-Hwan Lim, and Judith Fullerton. 2010. Recruitment of hard-to-reach population subgroups via adaptations of the snowball sampling strategy. *Nursing & Health Sciences* 12, 3 (9 2010), 369–374.

[82] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

[83] Vaibhav Saini, Hitesh Sajnani, and Cristina Lopes. 2018. Cloned and non-cloned Java methods: a comparative study. *Empirical Software Engineering* (2018), 1–47.

[84] Hitesh Sajnani, Vaibhav Saini, and Cristina V Lopes. 2014. A comparative study of bug patterns in java cloned and non-cloned code. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE.

[85] Hanan Samet. 1981. Experience with software conversion. *Software: Practice and Experience* 11, 10 (1981), 1053–1069.

[86] Walt Scacchi. 2007. Free/open source software development: recent research results and emerging opportunities. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

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

[88] Janet Siegmund, Christian Kästner, JÃűrg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19, 5 (10 2014), 1299–1334.

[89] Susan Elliott Sim, Charles LA Clarke, and Richard C Holt. 1998. Archetypal source code searches: A survey of software developers and maintainers. In *International Workshop on Program Comprehension (IWPC)*. IEEE.

[90] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.

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

[92] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium (USENIX Security)*.

[93] A.A. Terekhov. 2001. Automating language conversion: a case study (an extended abstract). *IEEE International Conference on Software Maintenance (ICSM)* (2001), 654–658.

[94] Andrey A. Terekhov and Chris Verhoef. 2000. The realities of language conversions. *IEEE Software* 17, 6 (2000), 111–124.

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

[96] Christoph Treude and Martin P Robillard. 2017. Understanding stack overflow code fragments. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[97] Iain Truskett. [n.d.]. Perl Regular Expressions Reference - Perl. https://perldoc.perl.org/5.22.0/perlreref.html.

[98] Brink Van Der Merwe, Nicolaas Weideman, and Martin Berglund. 2017. Turning Evil Regexes Harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists (SAICSIT)*.

[99] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. 2013. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *2013 International Conference on Social Computing*. IEEE, 188–195.

[100] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. *International Conference on Software Testing, Verification and Validation (ICST)* (2010).

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

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

[103] Richard Waters. 1988. Program translation via abstraction and reimplementation - Software Engineering. *IEEE Transactions on Software Engineering* 14, 8 (1988).

[104] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. 2016. 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)*, Vol. 9705. 322–334.

[105] Wikipedia contributors. 2018. Regular expression — Wikipedia, The Free Encyclopedia. https://web.archive.org/web/20180920152821/https://en.wikipedia.org/w/index.php?title=Regular_expression.

[106] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE)*. IEEE.

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

[108] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Mining Software Repositories (MSR)*. ACM, 391–402.

[109] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. 2017. Stack Overflow in Github: Any Snippets There?. In *IEEE International Working Conference on Mining Software Repositories (MSR)*.

[110] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Online Code Examples Reliable? An Empirical Study of API Misuse on Stack Overflow. In *International Conference on Software Engineering (ICSE)*.

[111] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *International Conference on Software Engineering*.