Testing Regex Generalizability And Its Implications
A Large-Scale Many-Language Measurement Study

James C. Davis, Daniel Moyer, and Ayaan M. Kazerouni
Department of Computer Science
Virginia Tech
{davisjam, dmoyer, ayaan}@vt.edu

Dongyoon Lee
Department of Computer Science
Stony Brook University and Virginia Tech
dongyoon@cs.stonybrook.edu

***Abstract*—The regular expression (regex) practices of software engineers affect the maintainability, correctness, and security of their software applications. Empirical research has described characteristics like the distribution of regex feature usage, the structural complexity of regexes, and worst-case regex match behaviors. But researchers have not critically examined the methodology they follow to extract regexes, and findings to date are typically generalized from regexes written in only 1–2 programming languages. This is an incomplete foundation.**

Generalizing existing research depends on validating two hypotheses: (1) Various regex extraction methodologies yield similar results, and (2) Regex characteristics are similar across programming languages. To test these hypotheses, we defined eight regex metrics to capture the dimensions of regex representation, string language diversity, and worst-case match complexity. We report that the two competing regex extraction methodologies yield comparable corpuses, suggesting that simpler regex extraction techniques will still yield sound corpuses. But in comparing regexes across programming languages, we found significant differences in some characteristics by programming language. Our findings have bearing on future empirical methodology, as the programming language should be considered, and generalizability will not be assured. Our measurements on a corpus of 537,806 regexes can guide data-driven designs of a new generation of regex tools and regex engines.

"There are more things in heaven and earth, Horatio,
Than are dreamt of in your philosophy."

–Hamlet

I. Introduction

For such a widely-used programming tool, regexes have been surprisingly understudied. Although they have been around for decades and appear in an estimated 30–40 % of software projects [1], [2], only recently have they been investigated from a software engineering perspective. Empirical studies have examined topics like regex readability [3], feature usage [1], structural complexity [4], evolution [5], and worst-case performance (and concomitant security vulnerabilities) [2], [6]. These studies have often been based on small-scale regex corpuses extracted from a few thousand software projects. Can their findings be generalized?

In this paper we formulate and test two generalizability hypotheses underlying prior research. *First*, we test whether prior results may have been biased by following different regex extraction methodologies (§V). Researchers have extracted regexes using static analysis [1], [2], [7] or runtime instrumentation [4], and to generalize from one methodology to the other we must show that the extracted regexes are similar. *Second*, we test whether prior results generalize to other programming languages. For this test we rely on a large-scale corpus of 537,806 regexes extracted from 193,524 projects across eight programming languages (§VI). Our generalization efforts depend on a comprehensive set of regex metrics (§IV).

Our findings support a nuanced notion of universal regex practices. In our first experiment, we show that the regex extraction methodology does not produce significantly different regex corpuses. In our second experiment, we found that the regexes from different programming languages are not significantly different on four of our eight metrics, and on the other metrics only a few languages are outliers. Because regexes appear to be similar across programming languages, we were able to replicate many findings from prior research in new programming languages on a larger regex corpus (§VII).

In §VIII we discuss the implications of our measurements for regex tool designers and regex engine developers. For example, visualization designers should ensure their approaches render well on realistically-sized regexes, and regex engine designers might prioritize regex feature support and optimizations based on their relative frequency of use in real regexes.

This work makes the following contributions:

- We identify two generalizability hypotheses underpinning existing empirical regex research (§III).
- We define a comprehensive set of regex metrics permitting regex characterization across three dimensions (§IV).
- We test these hypotheses using two regex corpuses collected from 75 K and 190 K software projects, respectively, written in three and eight popular programming languages. The generalizability hypotheses generally hold (sections V and VI).
- We test the replicability of prior regex research, and show that this research generalizes (§VII). We describe the potential effect of outliers on the findings of prior work.
- We discuss the implications of our measurements for regex tool designers and regex engine developers (§VIII).

II. Background

A. Regular Expressions, Automata, and ReDoS

A regex is a way to describe strings that match a certain pattern. Regexes are supported in most popular programming languages. Software engineers use them to concisely conduct sophisticated string operations like text processing, input sanitization, and code search [8], [9].

Programming languages implement regex matching using automaton simulation. A programming language's *regex engine* converts a regex pattern to a Non-deterministic Finite Automaton (NFA) or Deterministic Finite Automaton (DFA) representation. The regex engine tests for a pattern match by simulating the behavior of the automaton on a candidate string using static DFA simulation [10], Spencer's backtracking NFA simulation [11], or Thompson's dynamic DFA simulation [12]. Most regex features, like Kleene stars (`/a*/`) and custom character classes (`/[aeiou]/`), are truly regular in the automata-theoretic sense and can be modeled through the appropriate construction of the automaton graph and transition function. Some regex engines also support extended regex features like *backreferences* and *lookaround assertions*, which are non-regular and entail a more complex automaton representation and simulation [13]–[16].

The worst-case time complexity of a regex match varies widely by regex engine, exposing many applications to a denial of service vector. Most programming languages use Spencer's algorithm, which supports extended regex features but suffers from exponential worst-case time complexity [17], [18] due to the "catastrophic backtracking" that can occur while simulating NFAs with high ambiguity [6], [19], [20]. Super-linear worst-case regex match time can lead to an algorithmic complexity attack [21] known as Regular expression Denial of Service (ReDoS) [17], [22], [23]. In a ReDoS attack, an attacker triggers polynomial or exponential regex match behavior in server-side software to divert resources away from legitimate clients. For example, the company Cloudflare had an outage in July 2019 due to a super-linear regex match [24].

B. Recent Regex Research: Tools and Characteristics

Though regex research has historically focused on the mathematical properties of regexes, recent work has examined regexes from a software engineering perspective. Researchers have proposed a variety of tools to support engineers working with regexes. For example, regex visualizations have been proposed for comprehension [25]–[27]; regex input generators for regexes have been developed to support comprehension and testing [28]–[30]; and researchers have combated the ReDoS security threat through mechanisms for worst-case regex detection [6], [19], [31], [32] and prevention [18], [33]–[36].

The better we understand how and why software engineers use regexes, the better tools we can build to support them. To guide this endeavor, empirical regex researchers have sought to understand the characteristics of real-world regexes. The efforts of these researchers have provided many hints about how engineers use regexes in practice. Regexes are *widely used*, reportedly appearing in 30–40 % of software projects with applications like input sanitization, error checking, document rendering, linting, and unit testing [1], [2], [4]. Software engineers may rely more heavily on some *regex features* than others, possibly tied to the relative comprehensibility of different features [3]. Features like quantifiers, capture groups, and character classes are commonly used in Python, while backreferences and lookaround assertions rarely appear in practical regexes [1]. Engineers may *under-test* their regexes, perhaps relying on line coverage instead of automaton graph coverage [4]. Most regexes may go unmodified after entering version control [5]. And many prominent software modules and web services rely on *super-linear regexes* and are vulnerable to ReDoS [2], [6], [32].

If these preliminary empirical regex findings generalize, they can guide research into more fruitful directions and nip others in the bud [37]. For example, if regexes are as widely used as is thought, then visualization and input generation tools can be valuable aids for many developers. And if super-linear worst-case time complexity is as common as has been estimated, then addressing this behavior by overhauling regex engines seems natural. Conversely, if regexes do not change after entering version control [5], then regex-specific differencing tools (e.g., for code review) may not have great utility. And if non-regular regex extensions like backreferences and lookaround assertions are as rare universally as initial results suggest, then they should be a low priority for tool support and regex engine optimizations.

TABLE I
EXISTING REGEX CORPUSES. NO COMPARISON HAS BEEN MADE BETWEEN EXTRACTION METHODS. REGEX CHARACTERISTICS HAVE BEEN STUDIED CAREFULLY IN THREE PROGRAMMING LANGUAGES. *THIS CORPUS HAS ONLY BEEN USED TO COMPARE REGEX ENGINES (SEE §X).

Corpus	Extraction method	Languages (# Projects)
[1]	Static analysis	Python (4 K)
[2]	Static analysis	JavaScript (375 K), Python (72 K)
[4]	Program instrumentation	Java (1.2 K)
[7]*	Static analysis	Eight prog. languages (190 K)

III. Motivation: Assumptions and Applications

Empirical regex research depends on two generalizability hypotheses. Generalizing this research will permit us to guide future researchers and programming language designers.

A. Existing Regex Corpuses

As summarized in Table I, the corpuses used in prior empirical regex research were created using one of two regex extraction methodologies. The first three corpuses have been analyzed in terms of regex characteristics, covering only three programming languages. The fourth corpus has been used to compare regex engine behavior, but the characteristics of the regexes themselves have not been studied.

Comparing regex extraction methodologies. When a developer matches a string against a regex, they must specify the regex pattern and construct a Regex object. The regex pattern can be provided as a static string to the Regex constructor. Or the developer might wish to supply a variable string, e.g., to build a complex regex by concatenating its constituent parts. Rasool and Asif suggest that this practice of regex templating, which they call "abstract regexes," may make regexes easier to debug [38]. For a real-world example, the widely-used `marked` Markdown parser relies heavily on regex templating.[1] We illustrate these concepts in Listing 1.

Listing 1 Regex corpuses based on static analysis or program instrumentation may yield different results. For example, the regex used to match the `emailStr` is of varying complexity depending on the `regexType` flag. Static analysis might only be able to retrieve the simplest regex pattern, while an instrumented application or runtime might identify all three patterns if the software can be exercised thoroughly.

```
def isEmail(emailStr, regexType, externalRegex):
  if regexType == "SIMPLE_REGEX":
    reg = Regex(".+@.+")
  elif regexType == "COMPLEX_REGEX":
    NAME_REGEX = "[a-z0-9]+"
    DOMAIN_REGEX = "[a-z0-9]+(\.[a-z0-9]+)+"
    regex = NAME_REGEX + "@" + DOMAIN_REGEX
    reg = Regex(regex)
  else:
    reg = Regex(externalRegex)
  return reg.match(emailStr)
```

Regex corpuses have been constructed using either static analysis or program instrumentation (Table I). These approaches have familiar tradeoffs. Using static analysis, researchers can analyze an entire software project, but may not be able to extract dynamically defined regex patterns like those in Listing 1 without intra- and inter-function dataflow analysis. In contrast, runtime analysis can extract both statically and dynamically defined regex patterns so long as the relevant call sites are evaluated during execution. It is not clear whether a regex corpus based on one extraction methodology would be comparable to a regex corpus based on the other.

Regex variation by language? Regex research has provided hints about how engineers use regexes in practice, but these works have been isolated to practices in three programming languages. Engineers may choose a programming language based in part on their task [39], [40] ("the right tool for the job"), and some tasks may have greater call for pattern matching. It is not unreasonable to suppose that the characteristics of the regexes used to solve these problems may likewise vary by programming language.

B. Two Regex Generalizability Hypotheses

We wonder whether variations in regex extraction methodology, as well as the present restriction of regex analyses to only three programming languages, may mask variations in regex characteristics. We formulate these questions as two *regex generalizability hypotheses*: the Extraction Methodology (EM) and Cross-Language (CL) hypotheses.

H-EM It does not matter whether a regex corpus is constructed using static analysis or program instrumentation. At scale, using either extraction methodology will yield a corpus with similar distributions of regex metrics.

H-CL Regex characteristics are similar across programming language. The distributions of regex metrics will be similar for software from different programming languages.

[1] See https://github.com/markedjs/marked.

C. Application: Data-Driven Regex Engine Design

Programming language designers and regex engine developers have several regex matching algorithms to choose from, including Thompson's [12] and Spencer's [11]. While the pros and cons of these algorithms can be debated, some are noticeably more suitable than others on certain regexes, e.g., the well-known advantage of Thompson over Spencer engines on ambiguous regexes [18]. To the best of our knowledge, the regex engines in many programming languages were designed without considering the characteristics of real regexes.

We lack both a comprehensive set of metrics that engine designers should consider, and the measurements of real regexes to guide their designs. To fill this gap, in the next section we describe metrics that can indicate the relative costs of different approaches (§IV), and we proceed to measure real regexes and discuss the implications (§VIII).

IV. A Comprehensive Set of Regex Metrics

In this section we introduce our comprehensive collection of regex metrics (Table II). We selected metrics to characterize a regex in three dimensions: its representation, the diversity of the language it describes, and the complexity of various algorithms to solve its membership problem. These metrics fulfill two purposes. First, they include most regex metrics considered in prior research, allowing us to evaluate generalizability. Second, our metrics include those of particular interest to the developers of regex tools and regex engines. In testing these hypotheses, we characterize the largest extant regex corpus in support of data-driven tool and engine designs.

A. Metrics for Regex Representation

We measure the representation of a regex in terms of the pattern and its corresponding automata. The features and structural complexity of a regex may impact regex comprehension [3], affecting areas like code re-use and code review. These metrics may also influence the design of visualization tools ("Which features does my visualization need to support? How will typical regexes look in my visualization?").

A regex's pattern representation is the face it shows to engineers. Measures on the pattern representation give some sense of the impression an engineer has when examining the regex. We first measure the *length* of this representation in terms of the number of characters in the string encoding of the pattern. Then we measure its *Chapman feature vector* [1], counting the number of times each regex feature is used. For example, for the regex `/(a+)\1+/` we would compute a regex length of 7 and report two uses of the + feature and one use of the capture group and backreference features.

The pattern representation of an (automata-theoretic) regex corresponds to an NFA and DFA representation used by a regex engine to answer regex language membership queries. As we discuss during our analysis, measures of the automata

TABLE II

REGEX METRICS ORGANIZED BY REPRESENTATION, LANGUAGE DIVERSITY, AND WORST-CASE MATCH COMPLEXITY. THE FINAL COLUMN REFERENCES PREVIOUS STUDIES THAT MEASURE OR APPLY THIS METRIC. *: NO PRIOR SCIENTIFIC MEASUREMENTS.

Dimension	Metric	Description	Implications	Prior Studies
Representation	Pattern length	Characters in the regex (C# translation)	Length affects visualization, comprehension	*
	Feature vector sparseness	Number of distinct features used	More features: harder to comprehend	*
	# NFA vertices	Number of vertices in an epsilon-free NFA	Size affects visualization, comprehension	[4]
Lang. diversity	# simple paths	Num. of representative matching strings	Comprehension; Test suite size	[29] (basis)
Complexity	DFA blow-up	Ratio of DFA vertices to NFA vertices	Feasibility of static DFA-based algorithm	*
	Mismatch ambiguity	Worst-case match time for backtracking NFA simulation	Feasibility of Spencer's algorithm	[2], [6]
	Average outdegree density	Average completeness of outgoing edge set	Cost of Thompson's algorithm	*
	Has super-linear features	Whether regex relies on super-linear regex features (backreferences, lookaround assertions)	Unavoidable super-linear match complexity	[1], [2], [4]

complexity can inform the design of a regex engine. We apply a Thompson-style construction [12] to generate an (epsilon-free) NFA: a graph with vertices corresponding to NFA states connected by labeled edges indicating the character to consume to transition from one state to another.[2] We measure *the number of vertices in the NFA graph*.

B. Metrics for Regex Language Diversity

A regex pattern encodes a string language, i.e., the family of strings that its corresponding automaton will match. We measure the diversity of each regex's language with an eye to its testability. The larger and more diverse a regex's language, the larger the variety in the strings the regex accepts, and the more difficult it is to completely test and validate it.

We operationalize the notion of diversity by measuring *the size of a set of representative matching strings* for that language. We do this by measuring the number of distinct paths from the start state to the accept state that use each node at most once (i.e., the automaton graph's simple paths [42]). Each of these paths corresponds to a string in the language of the regex and is distinct in some way from each other path. In particular, for every optional node there is a simple path that does and does not take it; for every disjunction `/a|b/` there are separate sets of simple paths exploring each option. This family of strings is illustrated in Figure 1.[3]

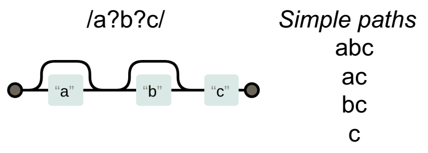

Fig. 1. Illustration of simple paths for the regex `/a?b?c/`.

[2]There are other NFA constructions optimized for fewer vertices or fewer edges, and a rich literature on the automata minimization problem [41]. We considered using minimized NFAs but found the algorithmic complexity was too great to handle the longer regexes in our corpus.

[3]This family can also be thought of as the (finite) set of strings in the language of the regex $r_{\text{loop-free}}$ after removing all loops from an original regex r. The size of this family can be determined recursively from the regex representation using rules like: $|characters|$ = 1; $|A*| = |A?| = |A\{0,\}| = |A| + 1$; $|A \vee B| = |A| + |B|$; $|AB| = |A| * |B|$.

Using simple paths to measure language diversity is similar in spirit to using basis paths as proposed by Larson and Kirk [29]. However, their goal was to obtain a manageable set of test strings. We believe succinctness comes at the cost of reduced comprehension. Basis paths can be used to ensure node coverage, but may not fully illustrate the range of "equivalence classes" in the regex's language the way that simple paths will.

C. Metrics for Regex Worst-Case Complexity

The worst-case time complexity of a regex match depends on the algorithm used to solve it, and several regex membership algorithms have been proposed with complexity ranging from linear to exponential [11], [12], [43]. Our metrics in this dimension can inform the design and application of regex engines based on different algorithms.

First, we consider algorithms that have super-linear worst-case complexity as a function of the regex (and input). Regex engines based on these algorithms are reportedly easier to implement and maintain [11], and so there is a tension between language designers' desires and the needs of software engineers who rely on "pathological" regexes in practice. If a high-complexity algorithm is used, pathological regexes become security liabilities — they can [2], [17] and have [36], [44] led to denial of service exploits (ReDoS).

Complexity in static DFA engines. One super-linear regex match algorithm statically converts the NFA to an equivalent DFA, offering linear time matches in the size of the input and the DFA. A DFA representation, however, is well known to have worst-case exponentially more states than its corresponding NFA representation [10]. If regexes with enormous DFA representations are common, this kind of algorithm is impractical; if they are rare, then it could be used alone or as the first approach in a hybrid regex engine.

To inform static DFA-based regex engines, we compute the following metric. Using the machinery from the representation metrics, we convert each regex NFA to a DFA. We compute *the ratio of DFA to NFA states* to evaluate how frequently this conversion results in an exponential state blow-up.

Complexity in Spencer engines. The Spencer algorithm [11] is a super-linear matching algorithm that relies on a backtracking-based NFA simulation. Spencer's algorithm is used in most programming languages, including JavaScript, Java, and Python [7], [18]. Each time this algorithm has a choice of edges, it takes one and saves the others to try later if the first path does not lead to a match. Several researchers have formalized the conditions for super-linear Spencer-style simulation due to NFA ambiguity [6], [19], [31], and shown that the worst-case simulation cost for a regex on a pathological input may be classified into linear, polynomial, or exponential as a function of the input string.

To inform Spencer-style regex engines, we compute the following metric. We measure a *regex's worst-case partial-match complexity in a Spencer-style engine*. For this measurement we use Weideman et al.'s analysis [20].[4] In our measurements we report the proportion of regexes that this analysis marks as polynomial and exponential among those it successfully analyzes. If super-linear regexes are common in software written in programming languages that use Spencer-style regex engines, the designers of those programming languages may wish to consider an alternative algorithm to reduce the risk of ReDoS vulnerabilities.

Complexity in Thompson engines. The Thompson algorithm [12], popularized by Cox [18], uses a dynamic-DFA based NFA simulation. It forms the basis of the Go and Rust regex engines. Each time a Thompson-style matching algorithm has a choice of edges, it simulates taking all of them, tracking the current set of possible NFA vertices and repeatedly computing the next set of vertices based on the available edges in the NFA transition table. In effect, a Thompson-style engine computes the DFA dynamically, not statically, and only computes the state-sets that are actually encountered on the input in question. It offers worst-case $O(n * m^2)$ complexity for inputs of length n and regexes with NFAs with m nodes, with the cost of each transition bounded by the number of outgoing edges that must be considered for each vertex in the current state-set. Note that each vertex may have outgoing edges to between zero and all m of the vertices in the graph, and the cost of each step of the Thompson algorithm depends on the number of outgoing edges from the current state-set.

We use the following metric to inform the design of a Thompson-style engine: *the average vertex outdegree density*, $\frac{1}{m}\sum_{1}^{m}\frac{\deg_{\text{vertex}_i}}{m} = \frac{|E|}{m^2}$, where E is the edge set of the automaton. This is a $[0,1]$ metric, 0 for completely unconnected graphs and 1 for completely connected graphs. For a Thompson-style engine, if the current state-set has x nodes, then it will cost an average of x times this metric to compute the next state-set.

Unavoidable super-linear complexity. Most regex engines support a feature set beyond traditional automata-theoretic regular expressions. Of particular note are backreferences, a self-referential construct proved to be worst-case exponential in the length of the input [45], and lookaround assertions, which are typically implemented with super-linear complexity. To round out our complexity metrics, we measure *the proportion of regexes that rely on these super-linear features*, through reference to the feature vector computed as part of the regex representation metrics. Understanding the popularity of these features may guide future regex engine developers in deciding whether or not to support these features. The most recent programming languages to gain mainstream adoption, Rust and Go, decided not to support these features, and it is not clear whether this decision will impose significant portability problems on engineers transitioning software from other languages to these ones.

D. Implementation of metric measurements

We built our measurement instruments on Microsoft's Automata library [46], which underlies the Rex regex input generation tool [28]. To the best of our knowledge this is the most advanced open-source regex manipulation library. Our fork extends the Automata library in several ways:

- We fixed several bugs in its automaton manipulations, eliminating long-running computation and memory exhaustion.
- We added support for generating the Chapman feature vector of a regex.
- We added support for collapsing certain expensive portions of a regex to facilitate simple path computation.
- We added support for emitting an automaton's graph in a format suitable for subsequent analysis.
- We introduced a command-line interface for automation.

The Automata library only supports .NET-compliant regexes. We therefore implemented an ad hoc syntactic regex conversion tool to translate regexes from other languages into a semantically equivalent .NET regex before measuring them. To reduce bias, we converted at least 95 % of the regexes originating in each language. These translations sufficed:

1) We replaced Python-style named capture groups and backreferences, `(?P<name>A)...(?P=<name>)`, with the .NET equivalent, `(?<name>A)...\k<name>`.
2) .NET only permits curly brackets to indicate repetition, while some other languages interpret curly brackets with non-numeric contents as a literal string. We escaped any curly bracket constructions of this form.
3) .NET does not support the `/\Q...\E/` escape notation. We removed the Q-E bookends and escaped the innards.
4) .NET does not support certain inline flags. We replaced the Unicode support flag with the "case insensitive" flag to preserve the presence of the feature while ensuring .NET compatibility.

The Automata library does not support simple path measurements, so we analyzed the NFA graph it produced using the NetworkX library [47].

The Automata library can parse all .NET-compliant regexes, but it can only produce NFAs for regexes that are regular (e.g., no support for backreferences, lookaround assertions, and greedy matches). We therefore omit automata measurements

[4]Weideman et al.'s analysis identifies both exponential and polynomial regex behavior and is open-source. Rathnayake and Thieleke's analysis only considers exponential behavior [19]; Wustholz et al.'s implementation is not open-source [6]; and Shen et al.'s mutation-based approach is unsound [32].

when necessary. We also omit automata measurements when the Automata library took more than 5 seconds to generate them.

V. The Extraction Methodology Hypothesis

Here we test the H-EM hypothesis: "It does not matter whether a regex corpus is constructed using static analysis or program instrumentation." *We found no reason to reject this hypothesis in the software we studied.*

We tested H-EM in the context of open-source software modules (libraries). Lacking access to closed-source software, we studied open-source software out of necessity. We opted to study modules rather than applications by choice. In our experience, modules have less variability in design and structure than do projects randomly sampled from GitHub, facilitating automated analysis. In addition, the ecosystem of most popular programming languages includes a large module registry, and so modules were a convenient target for our cross-language comparison experiment (H-CL). Using modules to test H-EM as well unified our methodology.

A. Methodology

Summary. Our methodology is summarized in Figure 2, and Table III provides the details. We targeted software modules written in the three most popular programming languages on GitHub: JavaScript, Java, and Python [48]. We extracted regexes using both static analysis and program instrumentation. After creating a regex corpus, we used statistical tests to determine whether there were significant differences between the regexes extracted using each methodology in any programming language.

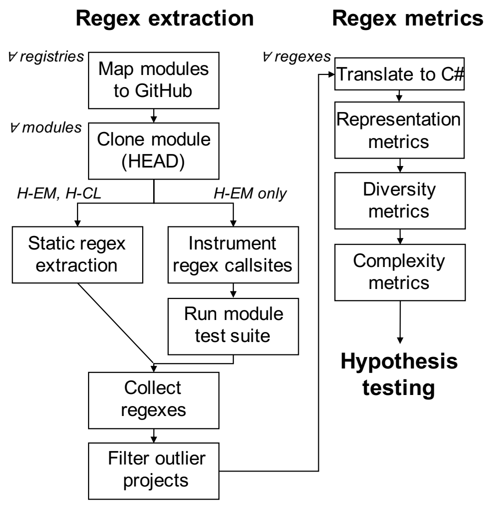

Fig. 2. Analysis flowchart. We performed regex extraction for H-EM, and for H-CL we leveraged an existing corpus derived using similar methodology.

Software. Modules were chosen by identifying the most prominent module registry for each language, mapping its modules to GitHub, and examining approximately the most important 25,000 modules from each. For JavaScript we used npm modules [49], for Python we used pypi modules [50], and for Java we used Maven modules [51]. Because software engineers commonly star modules that they depend on, we used a module's GitHub stars as a proxy for importance [52]. We extracted regexes from the entire module source code, both production code (e.g., `src/`) and test code (e.g., `test/`). We considered only source code written in the language appropriate for the module registry (e.g., only Python files for pypi modules, as determined by the `cloc` tool [53]).

Extraction through static analysis. We followed the methodology described in [1], [2], [7]. In each language, we used an AST builder to parse the module source code and visit the regex-creating call sites. We extracted statically-defined regex patterns from each such call site. We did not perform any dataflow analysis: we extracted string literals used as the regex pattern, and did not attempt to resolve non-literal arguments. For example, this extraction would only retrieve the "SIMPLE_REGEX" from Listing 1.

We examined the documentation for each programming language to learn the regex-creating call sites. Generally, regexes can be created directly through the language's Regex type and indirectly through methods on the language's String type. For example, in JavaScript you can create a regex directly using a regex literal, `/pattern/`, or the RegExp constructor, `new RegExp(pattern)`, or indirectly using a String method like `s.search(pattern)`. The AST libraries and regex-creating call sites we identified for each language are listed in Table III.

Extraction through program instrumentation. We followed a methodology similar to that of Wang and Stolee [4], but repaired one of their threats to validity. We targeted the same regex-creating call sites as we did in the static analysis. We applied a program transformation to instrument the (potentially variable) regex pattern argument at these call sites. Our instrumentation consisted of an inline anonymous function to log the pattern and return it, avoiding side effects. We then executed the test suites for the modules and collected the regexes that reached our instrumentation code. For example, this extraction would retrieve each of the regexes from Listing 1, provided the test suite covered each path.

We automatically executed the test suite for each module that used one of the common build systems for its registry (Table III). We identified these build systems using a mix of Internet searches and iterative analysis of modules from each registry. Because our source code-level instrumentation did not follow the coding conventions of the projects, some build attempts initially failed during an early linting stage. We configured our builds to skip or ignore the results of linting.

We found that many modules did not have test suites [65], and others failed to build due to external dependencies. We took several measures to increase the number of successful test executions. In Java, we installed all Android SDKs and Build Tools using Google's `sdkmanager`, permitting us to build many modules intended for use on Android. In Python, we attempted to run test suites under Python 2.7 and Python 3.5/6 using many different build systems. However, these ad

TABLE III
REGEX EXTRACTION DETAILS FOR H-EM. FOR STATIC ANALYSIS, WE EXTRACTED ANY CONSTANT REGEX PATTERNS USED AT THESE CALL SITES. FOR PROGRAM INSTRUMENTATION, WE WRAPPED THESE CALL SITES WITH A CALL TO A LOG ROUTINE.

Language	Regex call sites	AST module(s)	Build system(s)	Sample commands
JavaScript	*RegExp*: RegExp literals, RegExp constructor *String methods*: match, matchAll, search	Babel [54]	npm [55]	npm install-build-test
Java	*java.util.regex.Pattern*: compile, matches *String methods*: matches, replaceFirst, replaceAll, split	JavaParser [56]	Maven [57], Gradle [58]	mvn clean-compile-test
Python	*re module*: compile, escape, findall, finditer, fullmatch, match, search, split, sub, subn	ast, astor [59]	Distutils [60], Tox [61], Nox [62], Pytest [63], Nose [64]	python3 setup.py test

hoc approaches may have caused us to miss projects with other build systems or dependencies.

Collecting regexes via source code instrumentation ensured that we captured only the regexes created within each module, permitting direct comparison of the regexes extracted through the two different methodologies. This approach counters one of the threats to [4], which instrumented the language runtime and attempted to filter out third-party regexes.

Constructing the regex corpus. After extracting regexes from each module using the two methods, we combined the results into a corpus of unique regex patterns based on string equality of the regex pattern representations. We then noticed that some projects contributed orders of magnitude more regexes to the corpus than others did. The median number of unique regexes in regex-using projects was 1–3 in our experiment, while a few outlier libraries defined hundreds or thousands of distinct regexes — enough to bias statistical summaries of the regex corpus.[5] We therefore omitted regexes from projects at or above the 99th percentile of the number of unique regexes per project.

The regex corpus used to test the H-EM hypothesis is summarized in Table IV. Several elements of this corpus are worth noting. The corpus contains a moderate number of regexes extracted using static analysis and program instrumentation, ranging from around 15 K (Java) to around 80 K (JavaScript). We found that 30–50 % of the modules in each language used at least one regex, supporting previous estimates [1], [2], [4]. We were able to extract regexes from 3 K–4 K modules using program instrumentation, or about one third of the number from which we obtained regexes through static analysis.[6] Lastly, as you can see in the final row of Table IV, about half of the regexes obtained through program instrumentation were *not* obtained through static analysis and would thus not have been captured by a static-only extraction methodology.

Threats and considerations. Our approach is best-effort, neither sound nor complete. JavaScript and Python are dynamically typed, which could lead to *false positives* (non-regexes entering our corpus). For example, our JavaScript instrumentation relies on method names and signatures to find regexes, and for example may emit non-regexes if a class has a "match" method that shares the signature of the corresponding String method (Table III). Our analysis is also subject to *false negatives*, through modules that could not be parsed or built by our analyses (e.g., unsupported language versions or unfamiliar build systems), and through modules that create regexes via third-party APIs (e.g., using an "escape special chars and return a Regex" API). We appeal to the scale of our dataset to ameliorate concerns about corpus validity.

TABLE IV
SUMMARY OF CORPUS USED TO TEST H-EM. FOR EACH CELL "X (Y)", WE OBTAINED X UNIQUE REGEXES ACROSS (Y) REGEX-USING MODULES. THE FINAL ROW GIVES THE REGEX INTERSECTION. THIS CORPUS CONTAINS 124,800 UNIQUE REGEXES.

Extraction method	JavaScript	Java	Python
Static	71,799 (13.1 K)	10,237 (8.2 K)	27,641 (9.1 K)
Instrumentation	21,759 (4.4 K)	9,236 (3.1 K)	11,514 (3.7 K)
Static ∩ Inst.	13,633	3,463	5,690

For each module, we limited the static and dynamic phases of regex extraction to 10 minutes, and included in our corpus all regexes extracted during this time limit. Regex extraction and metric calculation were performed on a 10-node cluster of server-class nodes: Ubuntu 16.04, 48-core Intel Xeon E5-2650 CPU, 256 GB RAM.

B. Statistical Methods

We used statistical methods to determine whether there is evidence to reject H-EM — whether the regexes extracted using these two methodologies exhibited significant differences along any of our regex metrics. The statistical tests we chose were influenced by the distribution of the regex characteristics. Tests such as the Analysis of Variance (ANOVA) are typically used to evaluate such hypotheses. However, these tests require normality and homogeneity of variance, and none of the regex metric distributions met these assumptions. Therefore, we instead used the nonparametric Kruskal-Wallis test [66], with language and extraction mode as the treatment variables and our metrics as the dependent variables.

We found that hypothesis tests alone did not usefully describe our data. The scale of our regex corpus gave us tremendous statistical power, causing hypothesis tests to detect statistically significant but practically irrelevant differences in

[5]For example, the most prolific regex producers were pypi's `device_detector` module, which has 4,953 distinct regexes to match user-agent strings, and Maven's `recursive-expressions` module, which creates 3,398 regexes to test its extended regex APIs.

[6]We attribute this proportion to a combination of our failure to run the test suite, and poor code coverage within successful test suites.

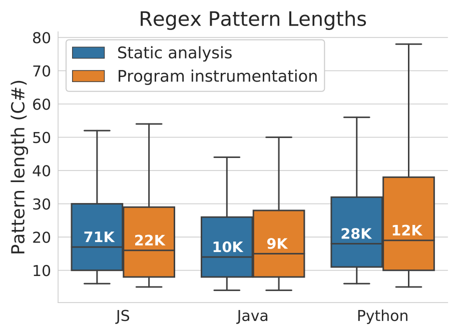

Fig. 3. Lengths of regexes extracted statically and dynamically, grouped by language. Whiskers indicate the (10, 90)th percentiles. Outliers are not shown. The text in each box shows the total number of regexes included in that group.

TABLE V
METRICS FOR EACH PROGRAMMING LANGUAGE IN THE H-CL EXPERIMENT. THE SECOND COLUMN GIVES THE RANGE OF THE MEDIAN OR THE OBSERVED PERCENTAGE, AND THE THIRD NOTES PROGRAMMING LANGUAGES WITH SIGNIFICANT DIFFERENCES FROM OTHER LANGUAGES.

Metric	Low / High	Unusual langs.
Length	Perl: 14 / Go: 21	Perl
Feat. vect. sparseness	Ruby: 2 / Go: 4	Ruby, JS
# NFA vertices	Java: 7 / Ruby 14	Ruby
# simple paths	Go: 1 / Python: 2	–
DFA blow-up	Perl 1.1 / Ruby 1.7	–
Mismatch ambiguity	Ruby: 19.1 % / Python: 38.4 %	–
Avg. outdegree density	Ruby: 0.08 / Java: 0.19	Ruby
Has super-linear features	Perl: 2.3 % / JS: 4.3 %	–

the data. So, after performing the Kruskal-Wallis hypothesis test, we calculated effect sizes for pairwise differences between groups. Because the distributions of regex characteristics do not meet the conditions assumed by parametric statistical tests, we applied a nonparametric difference effect size measurement d_r derived from the commonly used Cohen's d [67]. The d_r measure is a scaled robust estimator of Cohen's d proposed by Algina et al. [68], and shown to be robust to non-normal and non-homogeneous data [69]. It takes on scaled values indicating the size of the difference between two samples, ranging from 0 (no difference) to 1 (large difference).

C. Results

As indicated in Table IV, to test H-EM we split the regex corpus into two subsets: those extracted using static analysis, and those extracted using program instrumentation. Regexes found using both techniques were included in both subsets.

We compared the two subsets in terms of the metrics described in §IV. For all metrics, we found negligible-to-small effect sizes ($d_r <= 0.3$) between the static and dynamic subsets within each language. Figure 3 is illustrative: the similarity of regex lengths between the two subsets in each language is visually apparent. Other metrics look similar.

Therefore, we are unable to reject the null hypothesis H-EM. This conclusion supports the generalizability of prior empirical regex findings — from regexes declared using string literals to those generated dynamically, and vice versa.

VI. The Cross-Language Hypothesis

Here we test the H-CL hypothesis: "Regex characteristics are similar across programming languages." The H-CL hypothesis held for many characteristics. However, we identified several metrics on which there were moderate to large effect sizes between programming languages. *Not all regex characteristics span programming languages. Some differ significantly.*

A. Methodology

As we reported in §V, we did not reject the H-EM hypothesis in any of the three programming languages we studied. We used this finding as a basis for our methodology for testing H-CL. We evaluated the regex characteristics for software in many languages based solely on regexes obtained through static analysis. For this comparison, we drew on the polyglot regex corpus developed in [7]. This corpus contains 537,806 unique static regexes extracted from 193,524 popular software modules written in eight programming languages: JavaScript, Java, PHP, Python, Ruby, Go, Perl, and Rust. These regexes were obtained statically using extraction methods similar to those described in §V-A.

We followed the same measurement and statistical approach for H-CL that we did for H-EM. We measured the characteristics of the regexes in the polyglot regex corpus and again found that the distributions did not meet the conditions of normality and homogeneity of variance. Again the large sample size caused nonparametric Kruskal-Wallis hypothesis tests to yield uniformly significant differences. Thus, we report programming languages with a moderate ($d_r > 0.5$) or large ($d_r > 0.7$) pairwise effect size.

B. Results

Table V summarizes the results for each metric. We report the details for the metrics with significant effect sizes below. In §VIII we discuss some of the implications of these and other measurements.

- *Pattern length.* **Perl** regexes tend to be shorter than those in Go and Rust, with moderate effect sizes (Figure 4).
- *Features used.* Regexes in **Ruby** (large effects) and **JavaScript** (moderate) tend to use fewer features than regexes in PHP, Python, Go, and Rust (Figure 5).
- *# NFA vertices.* Regexes in **Ruby** tend to have more NFA vertices than those in Java and Perl (moderate) (Figure 6).
- *Average outdegree density.* Regexes in **Ruby** have a significantly smaller outdegree density than those in Perl, PHP, and Rust (moderate), and Java (large) (Figure 7).

VII. Replicating Previous Regex Research

Using the H-CL corpus, we attempted to replicate and generalize many of the findings described in §II-B.

Regex use. In agreement with prior estimates of regexes in 30–40 % of modules (Python, JavaScript [1], [2]), regex use is common in the modules that contributed to the H-CL

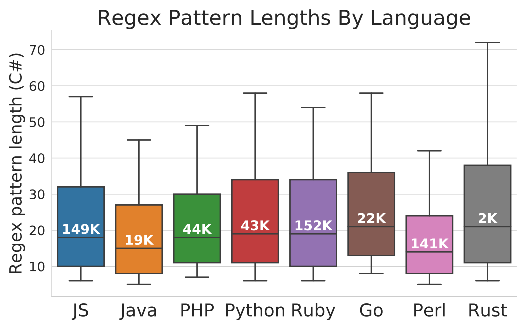

Fig. 4. Regex lengths per language. Whiskers are (10, 90)th percentiles. Outliers are not shown.

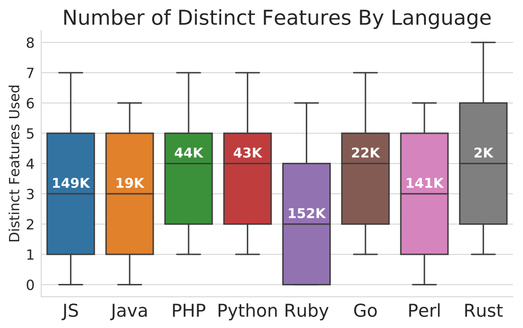

Fig. 5. Number of distinct features used by regexes in different programming languages. Whiskers indicate the (10, 90)th percentiles. Outliers are not shown.

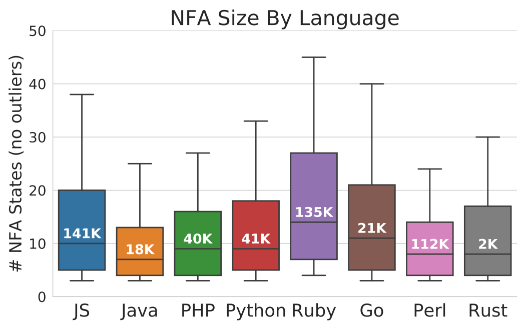

Fig. 6. Regex NFA size (# vertices) per language. Whiskers are (10, 90)th percentiles. Outliers are not shown.

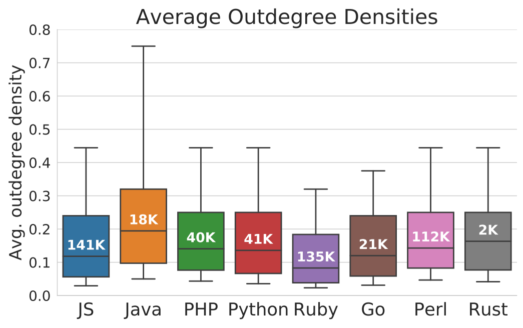

Fig. 7. Average outdegree density for each language. Whiskers are (10, 90)th percentiles. Outliers are not shown.

regex corpus, ranging from 23 % (Go) to 71 % (Perl). This finding did not generalize to **Rust**; only 5 % of Rust modules contained regexes.

Regex feature popularity. Feature usage rates in Python regexes were in agreement with findings from Chapman and Stolee [1]. The relative popularity ranking of different regex features is approximately similar across all programming languages in our corpus. For example, across languages, capture groups like `/(a)/` are a popular feature, while inline flag changes like `/(?i)CaSE/` are relatively rarely used.

Super-linear regexes. The frequency of super-linear behavior when applying partial regex matches in Spencer-style engines has been estimated at 20 % in Java [6] and around 10 % in JavaScript and Python [7], and with most of these matches exhibiting polynomial rather than exponential behavior. Our results agreed, estimating super-linear regex rates between 20–40 % under partial-match semantics, with a majority polynomial. These rates are upper bounds due to two causes of false positives. First, some of those regexes are used with full-match semantics, not partial-match semantics. Second, we did not dynamically each regex's performance in its programming language(s). Real Spencer-style regex engines may not meet all of the assumptions of Weideman et al.'s Java-centric performance analysis.

Prior researchers have reported that developers do not commonly use super-linear features (backreferences, lookaround assertions), with rates below 5 % reported in JavaScript, Python, and Java [2], [4]. This rate holds in all programming languages we studied that support those features.

Automaton sizes. We were not initially able to replicate findings from Wang and Stolee's work describing automaton sizes [4]. They reported that the (Java) regexes in their corpus, obtained using program instrumentation, had much larger DFAs than we found, with a 75th percentile of 70 nodes and 212 edges. The Java regexes in the corpus we analyzed (obtained using static analysis) have a 75th percentile of only 10 nodes and 70 edges. We first confirmed that our measurement instrument could replicate their results on their corpus. We then wondered if a few atypical projects might dominate their corpus, as in our corpus prior to our filtering step (§V-A). Indeed, we found that 19 source files in their corpus sat at or above the 99th percentile of unique regexes, and contributed more than half of the unique regexes in their corpus. After filtering out these files, our two corpuses had similar DFA measures.

This comparison emphasizes the importance of considering outlier projects during regex corpus construction. In both corpuses, a few projects contained enough regexes to bias the statistics derived from analyzing thousands of projects. We believe filtering out the regexes from these outlier projects offers a more accurate perspective on the population of "average" regex-using projects. However, this may be a matter of preference; perhaps major users of regexes deserve a greater voice in corpuses. We are grateful to Wang and Stolee's commitment to open science, permitting us to confirm that this phenomenon occurred in both sets of software and that the same filtering approach was effective on both sets.

VIII. Discussion

A. Implications of regex measurements

In the preceding sections we applied our measurements to test the validity of the H-EM and H-CL hypotheses. In those experiments we compared the *relative* values of the measures in different subsets of regex corpuses. However, the *specific* values of our measurements may be of interest to regex tool designers and regex engine developers. Though there are outliers in each category, appropriate percentiles are useful for reasoning about the common case of regexes encountered by regex tools and engines.

Regex Representation. Measures of regex representation (pattern, automaton) may be the most relevant for regex visualization and debugging tools. The (25,75)th percentile lengths of regexes in every language are between 5 and 40 characters, with medians of between 15 and 20 characters. Pattern-based regex tools (e.g., syntax highlighters [70], match/mismatch aids [71]) should be made to perform well on regexes of these lengths. Similarly, NFA-based regex tools (e.g., railroad diagrams [70]) should accommodate NFAs with between 5 and 30 NFA states, which will cover the (25,75)th percentile range in every language.

Language diversity. The 90th percentile of simple path family size for regexes in every language is at most 10. This means that the vast majority of regexes have at most ten simple paths through their NFA representation, so a covering set of at most ten inputs is sufficient to enumerate the "equivalence classes" of these regexes. Larson and Kirk's basis path-based approach [29] would yield even fewer inputs. Thus, exhaustive representative input generation is quite feasible for most regexes. This is not currently a feature in existing popular regex tools, and we recommend that they incorporate it as a cheap but potentially valuable feature.

Worst-case complexity. Our findings in this dimension can inform the design of the next generation of regex engines.

First, we report that *a static DFA-based matching algorithm is feasible for the vast majority of regexes* (Figure 8). The bottom 90 % of regexes have a blow-up factor of 2.5–3.75 in every language, implying that constructing and storing the DFA will not cost much more than the NFA would. A naive DFA approach would offer a guaranteed linear-time solution in the size of the original regex for 90 % of regexes.

Second, it appears that *super-linear regexes are common* in any programming language that uses a Spencer-style engine. Encouragingly, because we found that fewer than 5 % of regexes use super-linear features (backreferences, lookaround assertions) in any programming language, Thompson's algorithm can be applied to almost all regexes in every programming language. This change would address most ReDoS vulnerabilities in a single stroke. We therefore urge programming language designers to adopt hybrid regex engines, using Thompson's algorithm where possible and only relying on

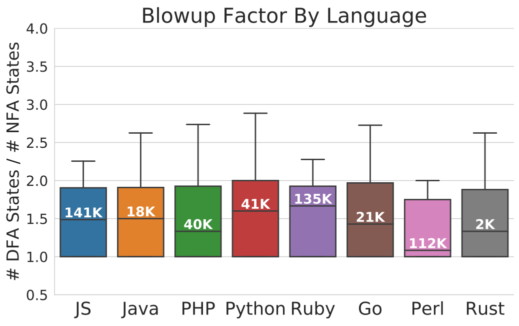

Fig. 8. DFA blowup for each language. Whiskers are (10, 90)th percentiles. Outliers are not shown.

Spencer-style algorithms in the rare cases when super-linear features are used. Eliminating support for super-linear features seems infeasible in languages that already support them, but a hybrid engine may be a viable solution to the ReDoS problem. This approach has previously been taken by `grep` [72].

Lastly, were regex engine designers to incorporate Thompson's algorithm, as have the designers of Rust and Go, they should consider its average cost. This cost depends on the number of transitions that must be considered as the algorithm updates its current state-set. In most programming languages the 90th percentile NFA outdegree density is no larger than that of the regexes in Rust (0.38) and Go (0.44), so lessons learned in Rust and Go may be applicable to other programming languages. However, in Java the 90th percentile NFA outdegree density is much higher, roughly 0.75. Thus, many languages can adopt a Thompson-style engine by referencing the approach in Rust and Go, but in Java more careful consideration may be required (Figure 7).

B. Thinking More Broadly

Software engineers must master many tools during their careers. These tools — including regular expressions, query languages (SQL, GraphQL), shell scripting (bash, PowerShell), version control systems (git, SVN), and virtualization technologies (Kubernetes, Docker) — are useful in and transferable to many engineering contexts. This paper is part of an effort to understand the ways in which software engineers use these external tools, the challenges they face as a result, and the ways in which researchers can support them. In this work we focused specifically on whether software engineers write similar regular expressions (regexes) in different programming languages. In concurrent work we have applied qualitative methods to enrich our understanding [73]. We anticipate that findings from one family of tools may generalize to others, and hope to merge our results with ongoing theory-building work on the tools and expertise that software engineers need [74].

IX. Threats to Validity

Internal validity. As noted in §V-A, our regex extraction methodologies were liable to both false positives (non-regexes included) and false negatives (real regexes excluded). Although regrettable, we do not believe that these inaccuracies systematically biased our corpus.

When the modules we studied accepted external regexes (e.g., the third case in Listing 1), our program instrumentation approach would capture any regexes specified through API calls in the test suite. These "test" regexes might resemble the other regexes in the module not because they would be similar in production usage, but because they were authored by the same developer(s) who wrote the other regexes in the module. Within a given module, the regexes extracted through static analysis and program instrumentation might have similar characteristics not because of intrinsic similarities but rather because of developer biases. We hope, however, that we sampled a diverse enough set of modules to observe many different developers' styles for regexes.

We considered all unique regexes equally, deduplicating the regexes by their pattern (string representation). Focusing on the characteristics of subsets of our corpus, e.g., popular regexes or test/production regexes, could be a topic for future study.

External validity. Part of the purpose of our study was to address threats to external validity in prior research, by testing whether the regex extraction methodology biased regex corpuses (§V) and whether previous empirical regex findings generalized to regexes written in other languages (§VI). We performed our experiments in the context of open-source software modules. The generalizability of this approach to other software — e.g., applications or closed-source software — is yet to be determined. Given the many programming languages considered in our analyses, we would be surprised if our findings did not generalize to regexes in other general-purpose programming languages. However, it is not clear whether they will apply to other pattern-matching contexts, e.g., to the regexes used in firewalls and intrusion detection systems [34], [75].

At each stage in our analysis (Figure 2), some regexes "leaked out". For example, we could not translate some regexes into the C# syntax. The losses were generally acceptable — for all but one of the metrics our measurements included at least 90 % of the regexes. The exception was the worst-case Spencer analysis, for which we could measure only about 80 % of the regexes. The missing regexes might have different characteristics, e.g., relying on unusual features.

Construct validity. Table II summarizes the metrics we used to characterize regexes. Most of these metrics, or their relatives, have been applied in prior work, and measure fundamental aspects of regexes. The new metrics we introduced are based on factors considered by existing regex engines.

We considered but omitted two metrics considered by prior work [2], [4], [7]. First, we do not generate mismatching strings for the language, although these may be of similar interest for testing purposes. These could be generated in a similar way by first taking the complement of the regex. Second, we do not attempt to label a regex based on the set of strings that it matches, e.g., "a regex for emails". The specific string language that a regex matches is an application concern. Our metrics are instead intended to characterize the components with which an engineer chose to construct a regex.

X. Related Work

Instead of automaton simulation, Brzozowski proposed using regex derivatives to implement regex matching [43]. We are not aware of any major regex engine that uses Brzozowski derivatives, so we did not discuss our metrics with regard to this algorithm.

We recently studied the problem of regex portability [7]. Where the current paper examines the differences between the *regexes* that developers use in different programming languages, in that work we explored the differences between the *regex engines* used by different programming languages. Both studies built on the same large-scale polyglot regex corpus.

XI. Conclusion

Previous empirical research on regex characteristics focused on statically-extracted regexes in software written in a small number of programming languages. This focus was not myopic: based on our suite of eight metrics we found that regex corpuses are similar whether they follow a regex extraction methodology based on static analysis or program instrumentation, and some characteristics of regexes are similar across many programming languages. However, some regex characteristics do not generalize across programming languages, and we encourage future empirical regex researchers to design their studies accordingly. We hope our methodological refinements and our efforts to validate generalizability hypotheses lay the foundation for further empirical regex research. We look forward to a new generation of regex tools and regex engines inspired by our measurements.

Reproducibility

To facilitate reproducibility, we published an artifact at https://doi.org/10.5281/zenodo.3424960. This artifact contains our measuring instruments for regex metrics (§IV) and the regex corpus used to test H-EM (§V). The many-language regex corpus used to test H-LR is available elsewhere [7].

Acknowledgments

We appreciate the remarks from the reviewers. We are grateful to C. Chapman, P. Wang, and K. Stolee for their feedback. This work was supported in part by the National Science Foundation grant CNS-1814430.

References

[1] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in Python," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

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

[3] C. Chapman, P. Wang, and K. T. Stolee, "Exploring Regular Expression Comprehension," in *ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[4] P. Wang and K. T. Stolee, "How well are regular expressions tested in the wild?" in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[5] P. Wang, G. R. Bai, and K. T. Stolee, "Exploring Regular Expression Evolution," in *Software Analysis, Evolution, and Reengineering (SANER)*, 2019.

[6] V. Wustholz, O. Olivo, M. J. H. Heule, and I. Dillig, "Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2017.

[7] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why arent regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[8] Wikipedia contributors, "Regular expression — Wikipedia, the free encyclopedia," https://web.archive.org/web/20180920152821/https://en.wikipedia.org/w/index.php?title=Regular_expression, 2018.

[9] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Centre for Advanced Studies on Collaborative Research (CASCON)*, 1997.

[10] M. Sipser, *Introduction to the Theory of Computation*. Thomson Course Technology Boston, 2006, vol. 2.

[11] H. Spencer, "A regular-expression matcher," in *Software solutions in C*, 1994, pp. 35–71.

[12] K. Thompson, "Regular Expression Search Algorithm," *Communications of the ACM (CACM)*, 1968.

[13] C. CÂMPEANU, K. SALOMAA, and S. YU, "A Formal Study of Practical Regular Expressions," *International Journal of Foundations of Computer Science*, vol. 14, no. 06, pp. 1007–1018, 2003.

[14] C. Câmpeanu and N. Santean, "On the intersection of regex languages with regular languages," *Theoretical Computer Science*, vol. 410, no. 24-25, pp. 2336–2344, 2009.

[15] M. Berglund and B. van der Merwe, "On the Semantics of Regular Expression parsing in the Wild," *Theoretical Computer Science*, vol. 578, pp. 292–304, 2015.

[16] M. Berglund, B. Van Der Merwe, B. Watson, and N. Weideman, "On the Semantics of Atomic Subgroups in Practical Regular Expressions," *Springer CIAA*, 2017.

[17] S. Crosby and T. H. E. U. Magazine, "Denial of service through regular expressions," in *USENIX Security work in progress report*, vol. 28, no. 6, 2003.

[18] R. Cox, "Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)," 2007. [Online]. Available: https://web.archive.org/web/20190906154019/https://swtch.com/~rsc/regexp/regexp1.html

[19] A. Rathnayake and H. Thielecke, "Static Analysis for Regular Expression Exponential Runtime via Substructural Logics," Tech. Rep., 2014.

[20] N. H. Weideman, "Static Analysis of Regular Expressions," Master's thesis, Stellenbosch University, 2017.

[21] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *USENIX Security*, 2003.

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

[23] B. Sullivan, "New Tool: SDL Regex Fuzzer," 2010. [Online]. Available: https://web.archive.org/web/20190917040133/https://www.microsoft.com/security/blog/2010/10/12/new-tool-sdl-regex-fuzzer//

[24] Graham-Cumming, John, "Details of the cloudflare outage on july 2, 2019," https://web.archive.org/web/20190712160002/https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[25] A. Blackwell, "SWYN: A visual representation for regular expressions," *Your Wish is My Command: Programming by ...*, pp. 1–18, 2001.

[26] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf, "RegViz: Visual Debugging of Regular Expressions," in *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion)*, 2014.

[27] J. Avallone, "Regexper," https://regexper.com/, 2013.

[28] M. Veanes, P. De Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," *International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[29] E. Larson and A. Kirk, "Generating Evil Test Strings for Regular Expressions," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[30] P. Arcaini, A. Gargantini, and E. Riccobene, "MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regular Expressions," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017.

[31] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson, "Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9705, 2016, pp. 322–334.

[32] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, "ReScue: Crafting Regular Expression DoS Attacks," in *ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[33] A. Hume, "A Tale of Two Greps," *Software - Practice and Experience*, vol. 18, no. 11, pp. 1063–1072, 1988.

[34] C. J. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of Snort," in *Proceedings DARPA Information Survivability Conference and Exposition II (DISCEX)*, 2001.

[35] B. Van Der Merwe, N. Weideman, and M. Berglund, "Turning Evil Regexes Harmless," in *SAICSIT*, 2017.

[36] J. C. Davis, E. R. Williamson, and D. Lee, "A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning," in *USENIX Security Symposium (USENIX Security)*, 2018.

[37] J. Brooks, Frederick P., "The Computer Scientist as Toolsmith II," *Communications of the ACM (CACM)*, vol. 39, no. 3, pp. 61–68, 1996.

[38] G. Rasool and N. Asif, "Software artifacts recovery using abstract regular expressions," in *IEEE International Multitopic Conference (INMIC)*, 2007.

[39] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[40] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[41] M. Holzer and M. Kutrib, "Descriptional and computational complexity of finite automata - A survey," *Information and Computation*, vol. 209, no. 3, pp. 456–470, 2011.

[42] "networkx.algorithms.simple_paths.all_simple_paths," https://web.archive.org/save/https://networkx.github.io/documentation/networkx-2.3/reference/algorithms/generated/networkx.algorithms.simple_paths.all_simple_paths.html.

[43] J. A. Brzozowski, "Derivatives of Regular Expressions," *Journal of the Association for Computing Machinery*, vol. 11, no. 4, pp. 481–494, 1964.

[44] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *USENIX Security Symposium (USENIX Security)*, 2018.

[45] A. V. Aho, *Algorithms for finding patterns in strings*. Elsevier, 1990, ch. 5, pp. 255–300.

[46] Microsoft, "Automata and transducer library for .net," https://github.com/AutomataDotNet/Automata.

[47] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[48] GitHub, "The state of the octoverse," https://octoverse.github.com/, 2018.

[49] "npm - the heart of the modern development community," https://www.npmjs.com/.

[50] "Pypi - the python package index," https://pypi.org/.

[51] "Maven repository," https://mvnrepository.com/.

[52] H. Borges and M. Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.
[53] "cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages," https://github.com/AlDanial/cloc.
[54] "Babel," https://babeljs.io/.
[55] "npm: A package manager for javascript," https://github.com/npm/cli.
[56] "Javaparser," https://javaparser.org/.
[57] "mvn: Apache maven," https://github.com/apache/maven.
[58] "gradle: Adaptable, fast automation for all," https://github.com/gradle/gradle.
[59] "astor: Python ast read/write," https://github.com/berkerpeksag/astor.
[60] "distutils: Building and installing python modules," https://docs.python.org/3/library/distutils.html.
[61] "tox: Command line driven ci frontend and development task automation tool," https://github.com/tox-dev/tox.
[62] "nox: Flexible test automation for python," https://github.com/theacodes/nox.
[63] "pytest: The pytest framework," https://github.com/pytest-dev/pytest.
[64] "nosetests: Nose is nicer testing for python," https://github.com/nose-devs/nose.
[65] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm," in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017.
[66] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.
[67] J. Cohen, "A power primer." *Psychological bulletin*, vol. 112, no. 1, p. 155, 1992.
[68] J. Algina, H. Keselman, and R. D. Penfield, "An alternative to cohen's standardized mean difference effect size: a robust parameter and confidence interval in the two independent groups case." *Psychological methods*, vol. 10, no. 3, p. 317, 2005.
[69] J. C.-H. Li, "Effect size measures in a two-independent-samples case with nonnormal and nonhomogeneous data," *Behavior Research Methods*, vol. 48, no. 4, pp. 1560–1574, Dec 2016.
[70] "Regexr: Learn, build, & test regex," https://regexr.com.
[71] "Online regex tester and debugger: Php, pcre, python, golang and javascript," https://regex101.com.
[72] J. E. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2002.
[73] L. G. Michael IV, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions," in *ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2019.
[74] S. Baltes and S. Diehl, "Towards a theory of software development expertise," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 187–200.
[75] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Large Installation System Administration Conference (LISA)*, 1999.