Abstract—The regular expression (regex) practices of software engineers are a recent area of interest. So far, empirical research has described characteristics like the distribution of regex feature usage, the structural complexity of regexes, and worst-case regex match behaviors. But most work to date has relied solely on statically-extracted regexes, and each study has usually examined only one programming language. This is an incomplete foundation on which to base new regex tools and engines.

Generalizing existing research depends on validating two hypotheses: (1) Static regexes are representative of all regexes; 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 then evaluated two regex corpuses to compare the characteristics of (1) static and dynamic regexes in three popular programming languages, and (2) static regexes across eight programming languages.

We report that static and dynamic regexes in a given programming language look alike after removing outliers, suggesting that a static-only regex corpus will generalize to all typical regexes within a programming language. 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 programming language should be considered and generalizability will not be assured. And our measurements on a corpus of 537,806 regexes can guide data-driven designs of a new generation of regex tools and regex engines.

I. INTRODUCTION

Regular expressions (regexes) are a programming language primitive in all popular programming languages. Software engineers use regexes to concisely conduct sophisticated string operations like text processing, input sanitization, and code search [1], [2]. Although regexes have been a programming language primitive for decades, only recently have they been scientifically investigated from a software engineering perspective. Empirical studies have examined topics like regex readability [3], feature usage [4], structural complexity [5], evolution [6], and worst-case performance (and concomitant security vulnerabilities) [7], [8].

Empirical software engineering research has taught us much about the characteristics of real regexes. But empirical software engineering shares the properties of any other experimental discipline. We hypothesize, sample, measure, and analyze. Then we generalize from our sample to a larger population. However, generalization takes care — it requires understanding how well the sample represents the population, and sometimes this relies on hypotheses about the relationship between the sample and the population.

In this paper we test two generalizability hypotheses underpinning empirical regular expression research. Research to date has built predominantly on two regex corpuses, both of approximately 10,000 regexes extracted from a few thousand projects written in one language (Python [4], Java [5]). But of course there are millions of projects written in dozens of programming languages, all of which may use regexes differently. We wanted to determine the extent to which existing results generalize both to their original populations of interest and to new ones. We therefore evaluate generalizability in two directions: the type of regexes (statically or dynamically constructed), and the programming language in which the regex is used. To accomplish these goals we define a comprehensive set of regex metrics (§IV) and perform measurements on a corpus of 537,806 regexes taken from 193,524 projects written in 8 programming languages.

Encouragingly, research on static regexes appears to generalize to dynamic regexes (§V). We found no significant differences between regexes whether they were statically or dynamically extracted from the software being studied. Furthermore, many preliminary findings — performed on a small number of projects in 1-2 programming languages — generalize to ~25K projects from each of 8 languages (§VI). However, we did identify several languages, most commonly Ruby, in which regexes were unusual along various metrics, giving nuance to notions of universal regex practices. In §VII 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 characterization across three critical dimensions (§IV).
• We test these hypotheses using two regex corpuses collected from 75K and 200K software projects, respectively, written in 3 and 8 popular programming languages. We report that some regex results generalize but others do not, and describe an important methodological refinement on prior work.
• We discuss the implications of our measurements for regex
tool designers and regex engine developers. We emphasize the importance of revisiting existing insecure regex engine designs and give guidance for a data-driven engine redesign.

II. BACKGROUND AND RELATED WORK

A. Regular Expressions, Automata, and ReDoS

A regular expression (regex) is a way to describe strings that follow a certain pattern. Regexes are supported in most popular programming languages, and are commonly used to solve problems such as input validation and find/replace [1].

Regex matching is commonly implemented through 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 [9], Spencer’s backtracking NFA simulation [10], or Thompson’s NFA-to-DFA simulation [11].

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 can readily support extended regex features but which suffers from exponential worst-case time complexity [12], [13] due to the “catastrophic backtracking” that can occur while simulating NFAs with high ambiguity [7], [14], [15]. This super-linear worst-case regex match behavior can be leveraged in an algorithmic complexity attack [16] known as Regular Expression Denial of Service (ReDoS) [12], [17], [18], in which an attacker exploits polynomial or exponential regex match behavior in server-side software to divert resources away from legitimate clients.

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 [19]–[21]; regex input generators for regexes have been developed to support comprehension and testing [22]–[24]; and researchers have combated the ReDoS security threat through worst-case regex detection [7], [14], [25], [26] and prevention [13], [27]–[30] mechanisms.

To guide the development of regex tools and engines, empirical regex researchers have sought to understand real-world regex characteristics. 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 as varied as input sanitization, error checking, and unit testing [4], [5], [8]. 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 [4]. Regexes automata are relatively large, and engineers may under-test regexes, perhaps relying on line coverage instead of automaton graph coverage [5]. Many prominent software modules and web services use super-linear regexes and are vulnerable to ReDoS [7], [8], [26]. Finally, most regexes may go unmodified after entering version control [6].

If these preliminary empirical regex studies generalize, they can guide research into more fruitful directions and nip others in the bud [31]. 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 [6], then regex-specific differencing tools (e.g., for code review) may not have great utility. And if non-“regular” [9] 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.

III. MOTIVATION: ASSUMPTIONS AND GAPS

A. The (Untested) Regex Generalizability Hypotheses

We believe that two critical hypotheses must be tested before accepting current empirical regex research with confidence. As summarized in Table I, the corpuses underlying prior empirical regex research are isolated to one type of regex (static or dynamic) and consider only three programming languages. But developers can also dynamically generate regexes, e.g., to construct more complex regexes from templates, and may have different regex needs (and practices) in different programming languages. We believe that the present restriction of regex corpuses to only three programming languages, as well as the lack of comparisons between even those languages, may mask distinctions in regex characteristics.

We formulate the assumptions underlying current empirical regex research in two regex generalizability hypotheses: the Static Regexes (SR) and Cross-Language (CL) hypotheses.

**H-SR** Static regexes are representative of all regexes. For software written in a given programming language, the distributions of regex metrics for static and dynamic regexes will be similar.

**H-CL** Regex characteristics span programming languages. For software written in different programming languages, the distributions of regex metrics will be similar.

<table>
<thead>
<tr>
<th>Publication</th>
<th>Regex Type</th>
<th>Languages (# Projects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4]</td>
<td>Static</td>
<td>Python (4K)</td>
</tr>
<tr>
<td>[8]</td>
<td>Static</td>
<td>JavaScript (375K), Python (72K)</td>
</tr>
<tr>
<td>[5]</td>
<td>Dynamic</td>
<td>Java (1.2K)</td>
</tr>
</tbody>
</table>
It is not clear what inferences can be drawn from existing regex corpuses until we have tested these hypotheses.

B. Gaps: Data-Driven Regex Engine Design

Programming language designers and regex engine developers have several regex matching algorithms to choose from, including Thompson’s [11] and Spencer’s [10]. While the pros and cons of these algorithms can be debated, some are noticeably more suitable than others on some regexes, e.g., the well-known advantage of Thompson over Spencer engines on regexes with high ambiguity [13]. But to the best of our knowledge, the regex engines built into many popular 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 (§VII).

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 its membership problem. These metrics fulfill two purposes. First, they include most regex metrics considered in prior research, and will thus allow 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 corpus of real regexes in support of data-driven tool and engine designs. Next we discuss the metrics for each dimension in detail.

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, code review, and regex “diff”erence generation (e.g., for 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?”).

The pattern representation is the face a regex 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. Then we measure its Chapman feature vector [4], counting the number of times an engineer chose to use each regex feature (e.g., a Kleene star or a backreference) to encode the regex.

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 will discuss during our analysis, measures of the automata complexity can inform the design of a regex engine.

We apply the Thompson construction [11] 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. We measure the number of vertices in the NFA graph.

B. Metrics for Regex Language Diversity

Regexes describe a language, i.e., the family of strings that its corresponding automaton will match. We measure the diversity of each regex’s language. These measures impact regex 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 simple paths of the graph [33]). Each of these paths corresponds to a string in the language of the regex and is distinct in some way from each other path: for every optional node there is a path that does and does not take it; for every disjunction /a | b/ there are separate sets of paths exploring each option. This family of strings is illustrated in Figure 1. Alternatively, it can 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 \( r \).

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.

Using simple paths is similar in spirit to that of Larson and Kirk, who computed linearly independent inputs (basis paths; cyclomatic complexity [34]) instead [23]. Their goal was to obtain a manageable set of test strings. But in reducing the string family, we believe they do not fully represent possible inputs the way simple paths do, potentially trading comprehension for succinctness. Basis paths can be used to ensure node coverage, but will not fully illustrate the range of “equivalence classes” in the regex’s language the way that simple paths will.

Fig. 1. Illustration of simple paths for the regex /\^a?b?c$/.

There are other NFA constructions optimized for fewer vertices or fewer edges, and a rich literature on the automata minimization problem [32]. The Thompson construction is, however, both well-known and efficient to implement. We considered using minimized NFAs but found the algorithmic complexity was too great to handle the longer regexes in our corpus. The size of the family of representative strings can be determined recursively from the regex representation using these rules: |characters\rangle = 1; \( |A\times| = |A^2| = |A\{0,1\}| = |A|+1; |A \lor B| = |A| + |B|; |AB| = |A| \times |B|.

3
Several researchers have formalized the conditions for super-linear Spencer-style simulation due to NFA ambiguity [7], [14], [25], and shown that the worst-case simulation cost for a regex on a pathological input may be classified into linear, polynomial, or exponential.

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 using Weideman et al.’s analysis [15]. In our measurements we report the proportion of regexes that this analysis reports to be super-linear among those it successfully analyzes. If super-linear regexes are common in software written in programming languages that use Spencer-style regex engines, these languages may wish to consider an alternative algorithm to reduce the risk of ReDoS vulnerabilities.

### Complexity in Thompson engines.

The Thompson algorithm [11], popularized by Cox [13], 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 NFA vertices in which it might be 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 $O(n \times m)$ 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,

$$\frac{1}{m} \sum_{v \in V} \frac{\text{deg}(v)}{m} = \frac{E}{m^2}.$$ 

This is a $[0,1]$ metric, 0 for completely unconnected graphs and 1 for completely connected graphs.

---

#### TABLE II

**Metrics characterizing regexes 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.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Metric</th>
<th>Description</th>
<th>Implications</th>
<th>Prior Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Representation</strong></td>
<td>Pattern length</td>
<td>Characters in the regex (C# translation)</td>
<td>Length affects visualization, comprehension</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feature vector sparseness</td>
<td>Number of distinct features used</td>
<td>More features: harder to comprehend</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td># NFA vertices</td>
<td>Number of vertices in an epsilon-free NFA</td>
<td>Size affects visualization, comprehension</td>
<td>[5]</td>
</tr>
<tr>
<td><strong>Lang. diversity</strong></td>
<td># simple paths</td>
<td>Num. of representative matching strings</td>
<td>Comprehension; Test suite size</td>
<td>[23] (basis)</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>DFA blow-up</td>
<td>Ratio of DFA vertices to NFA vertices</td>
<td>Feasibility of static DFA-based algorithm</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Mismatch ambiguity</td>
<td>Worst-case match time for backtracking NFA simulation</td>
<td>Feasibility of Spencer’s algorithm</td>
<td>[7], [8]</td>
</tr>
<tr>
<td></td>
<td>Average outdegree density</td>
<td>Average completeness of outgoing edge set</td>
<td>Cost of Thompson’s algorithm</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Has super-linear features</td>
<td>Whether regex relies on super-linear regex features (backrefs., lookarounds)</td>
<td>Unavoidable super-linear match complexity</td>
<td>[4], [5], [8]</td>
</tr>
</tbody>
</table>

---

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 [10], [11], [35]. 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 [10], 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 [8], [12] and have [30], [36] led to denial of service exploits (ReDoS).

**Complexity in static DFA engines.** One such super-linear matching 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 [9]. 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 into 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 [10] is a popular super-linear matching algorithm that relies on a backtracking-based NFA simulation. In concurrent work [37] we showed that it is used in most programming languages, including JavaScript, Java, and Python. Each time a Spencer-style matching 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. If the NFA is highly worst-case ambiguous [7], [25], *i.e.*, there is an input for which many different paths must be attempted, then the cost of Spencer-style simulation will be super-linear in the length of that input.
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 [38], and lookahead 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.

**D. Omitted metrics**

We do not include what might be considered a natural regex metric, viz. some characterization of the set of strings that it matches. We feel that the specific language that the regex matches is an application concern. Whether a given regex matches emails or numbers strikes us as irrelevant to most research perspectives, excepting efforts to categorize the purposes for which developers use regexes [4], [8]. For research purposes, we think what matters is not the specific set of strings matched by a regex but rather how a developer chooses to construct the regex.

**E. Implementation of metric measurements**

We built our primary measurement tool on Microsoft’s Automata library [39].\(^4\) Our fork extends the Automata library in several ways:

- We fixed several bugs in the built-in automaton manipulations, eliminating long-running computation and memory exhaustion.
- We added support for generating the feature vector of a regex during the parsing stage.
- 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, \((?P<\text{name}>A)\ldots(?P=\text{name}>),\) with the .NET equivalent, \((?:\text{name}>A)\ldots\text{k<name>}.\)
2. .NET only permits curly brackets to indicate repetition, while some other languages interpret curly brackets with non-numeric contents as a literal string\(^5\). We escaped any curly bracket constructions of this form.
3. .NET does not support the \(/\{Q\ldots\}E/\) 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 [41].

The Automata library only produces NFAs for regexes that are regular (i.e., no backreferences) and does not support some other features (e.g., lookahead assertions, greedy match). We therefore include feature vector measurements for all .NET-compliant regexes but omit automata measurements when necessary. We also omitted automata measurements when the Automata library took more than 5 seconds to generate them.

**V. H-SR: The Static Regex Hypothesis**

Here we test the H-SR hypothesis: “Static regexes are representative of all regexes.” We found no reason to reject this hypothesis in the software we studied.

We tested H-SR by examining the regexes used in 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, because the ecosystem of most popular programming languages includes a large module registry, modules are a convenient target for cross-language comparisons (H-CL), so using modules to test H-SR as well unified our methodology.

**A. Methodology**

Figure 2 summarizes our methodology. We collected the statically declared and dynamically created regexes from software modules written in the three most popular programming languages on GitHub: JavaScript, Python, and Java [42]. These are also the only languages in which prior empirical regex research has been conducted (Table I). After measuring the regexes, we used statistical tests to determine whether there were significant differences between the static and dynamic regexes used in each programming language.

**Software.** Modules were chosen by identifying the most prominent module registry for each language, mapping its modules to GitHub, and examining approximately the most

\(^4\)This is the library underlying the Rex regex input generation tool [22].

\(^5\)This syntax facilitates templating a regex into constituent parts, e.g., dynamically constructing a regex from the template \(/(\text{USER_RE})@/\text{DOMAIN_RE}/\) [40].
important 25,000 modules from each. For JavaScript we used npm [43] modules, for Python we used pypi [44] modules, and for Java we used Maven [45] modules. Because software engineers commonly star modules that they depend on [46], we used a module’s GitHub stars as a proxy for importance.

**Static/Dynamic RegEx Corpus.** To collect static and dynamic regexes, we used abstract syntax tree (AST) tools for each language to find all regex-creating call sites in the source code of each module, considering both “production” (e.g., src) and “test” code (e.g., test)\(^6\). We then extracted static regexes directly by walking the ASTs. For dynamic regexes, we replaced each regex expression in the source code with a small wrapper to log the regex during execution. We then executed the test suites for the modules and collected any regexes that were dynamically created during the test process.

**Static regex extraction / Instrument regex callsites.** 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 calls we identified for each language are listed in Table III. We believe this is the complete set of regex creation mechanisms in these languages.

In each language, we used an AST builder to parse the module source code and find these regex-creating call sites. During an AST walk, at each such call site we extract statically-declared regexes. We do not perform any dataflow analysis; we extract string literals used as the regex pattern, and do not attempt to resolve any non-literal arguments. We then rewrite the call site to emit the (potentially dynamic) regex argument during execution. To ensure that our instrumentation preserved the behavior of the original code, we replaced every regex argument with a call to an inline anonymous function that logged the regex and name of the source file and then returned the regex. The AST modules and instrumentation expressions we used in each language are summarized in Table III.

Collecting dynamic regexes via source code instrumentation ensures that we captured only the native regexes used in each module, permitting direct comparison of the static and dynamic regexes in this corpus. We thus counter one of the threats to validity of [5], which instrumented the language runtime instead and attempted to filter out third-party regexes.

**Run module test suite.** We automatically executed each module’s test suite, provided it used one of the common build systems for that 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 project, 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, 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 hoc approaches may have caused us to miss projects with other build systems or dependencies.

**Filter outlier projects.** The purpose of our experiment is to understand the characteristics of typical regexes. If a project’s use of regexes is atypical, then the characteristics of its regexes may also be atypical. We wish to omit such projects, and operationalize this by filtering out the regexes from projects that have an unusually large number of unique regexes. The median number of unique dynamic regexes in regex-using projects was 1-3 in these programming languages, while a few libraries defined hundreds or thousands of distinct regexes — enough to bias statistical summaries of the regex corpus. For example, the most prolific regex producers were pypi’s device_detector\(^7\), which has 4,953 distinct regexes to match user-agent strings, and Maven’s recursive-expressions\(^8\), which created 3,398 throwaway regexes to test its extended regex APIs. We therefore omitted regexes from projects that fell at or above the 99\(^{th}\) percentile of the number of unique regexes.

**Threats and considerations.** Our approach is best-effort, neither sound or complete. JavaScript and Python are dynamically typed, which could lead to false positives (non-regexes entering our corpus) of two forms. First, our JavaScript

---

\(^6\)We restricted our analysis to source code written in the language(s) used in the registry. For example, we did not instrument the Perl scripts that occasionally crop up in Python modules. We identified the language of each file using the cloc tool [47].

\(^7\)See https://www.github.com/thinkwelltwd/device_detector.

\(^8\)See https://github.com/ttulka/recursive-expressions.
suite, and poor code coverage within successful test suites.

Tests such as the Analysis of Variance (ANOVA) are typically influenced by the distribution of the regex characteristics. The statistical tests we chose to assess whether there are significant differences between the characteristics of statically extracted and dynamically extracted regexes. The statistical tests were chosen based on 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 regexes were influenced by the distribution of the regex characteristics.

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Regex call sites</th>
<th>AST modules</th>
<th>Build system(s)</th>
<th>Sample commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>re module: compile, escape, findall, finditer, fullmatch, match, search, split, sub, subn</td>
<td>ast, astore [53]</td>
<td>Distutils [54], Tox [55], Nox [56], Pytest [57], Nose [58]</td>
<td>python3 setup.py test</td>
</tr>
</tbody>
</table>

We used statistical methods to determine whether there is evidence to reject H-SR — whether there are significant differences between the characteristics of statically extracted and dynamically extracted regexes. 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 regexes were influenced by the distribution of the regex characteristics.

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, 256GB RAM.

**Resulting static/dynamic corpus.** The static/dynamic regex corpus used to test the H-SR hypothesis is summarized in Table IV. Several elements of the static/dynamic corpus are worth noting. We have a large corpus of static and dynamic regexes in each language, ranging from around 20K (Java) to around 90K (JavaScript). We found that 30-50% of the modules in each language used at least one regex, supporting previous estimates [4], [5], [8]. We were able to dynamically extract regexes from 3K-4K modules, or about one third of those that contained statically-declared regexes. The intersection of regexes is about half of the dynamic regexes in each language, meaning that half of these dynamically extracted regexes were not obtained statically and would thus not have been represented in a static-only analysis.

### B. Statistical Methods

We used statistical methods to determine whether there is evidence to reject H-SR — whether there are significant differences between the characteristics of statically extracted and dynamically extracted regexes. 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 regexes were influenced by the distribution of the regex characteristics.

As indicated in Table IV, to test H-SR we split the regex corpus into two subsets: 1) Regexes that were extracted statically, and 2) Regexes that were extracted dynamically. Any regexes found through both techniques are included in both subsets. In each subset we treat each regex equally, not weighted by how frequently they appeared (i.e., no duplicates).

We compared the two subsets in terms of the metrics described in §IV. For all metrics, we found negligible-to-small effect sizes ($d_r \leq 0.3$) between the static and dynamic subsets within each language. Figure 3 is illustrative: the similarity between 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-SR, supporting the generalizability of prior empirical regex findings — from static regexes to dynamic ones and vice versa.

VI. H-CL: THE CROSS-LANGUAGE HYPOTHESIS

Next we tested H-CL: “Regex characteristics span 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 find compelling evidence to reject the H-SR hypothesis in any of the three languages we studied. We used this finding as a basis for our methodology for testing H-CL, evaluating the regex characteristics for software in many languages based solely on statically-extracted regexes. For this comparison, we drew on a polyglot regex corpus developed in concurrent work [37]. This corpus contains 537,806 unique static regexes extracted from 193,524 popular software modules written in 8 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 as we did for H-SR. 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 §VII-B we discuss some of the implications of these and other measurements.

![Regex Pattern Lengths](image)

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.

- **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. DISCUSSION

A. Generalizing previous regex research

Using the H-CL corpus, we were able to generalize most of the empirical regex findings described in §II-B.

- **Regex use.** In agreement with prior estimates of regexes in 30-40% of modules (Python, JavaScript [4, 8]), regex use is common in the modules that contributed to the H-CL 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 [4]. The relative popularity ranking of different regex features is approximately similar across all programming languages in our corpus. For example, across languages, CG (capture groups) is a highly popular feature and OPT (options wrapper) is relatively rarely used.

    - **Super-linear regexes.** First, the frequency of super-linear behavior when applying partial regex matches in Spencer-style engines has been estimated at 20% in Java [7] and around 10% in JavaScript and Python [37], and with most of these matches exhibiting polynomial rather than exponential behavior. Our results agreed, estimating super-linear regex rates between 20-40%, with a majority polynomial. These rates are upper bounds because some of those regexes are used with full-match semantics, and because we did not dynamically test them in the language of origin. Second, 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 [5, 8]. This rate
Fig. 4. Regex lengths per language. Whiskers are (10, 90)\textsuperscript{th} percentiles. Outliers are not shown.

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

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

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

holds in all languages we studied in which those features are supported (i.e., excepting Go and Rust).

\textbf{Automaton sizes}. We were not initially able to replicate findings from Wang and Stolee’s work describing automaton sizes [5]. They reported that the (dynamic Java) regexes in their corpus have much larger DFAs than we found, with a 75\textsuperscript{th} percentile of 70 nodes and 212 edges. The (static Java) regexes in our corpus have a 75\textsuperscript{th} percentile of only 10 nodes and 70 edges. We first confirmed that our differing metric implementations (based respectively on the Automata and RE2 libraries) yielded similar results on similar regexes. We then wondered if a few atypical projects might dominate their corpus, as in our dynamic corpuses were prior to our filtering step (§V-A). Indeed, we found that 19 source files in their corpus sat at or above the 99\textsuperscript{th} 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 potential importance of the refinement we made in our own regex extraction methodology: filtering outlier projects. Some projects to dynamically generate enough regexes to bias a corpus derived from thousands of projects. We believe filtering out the regexes from these outlier projects is necessary to obtain an accurate perspective on the population of “average” regex-using projects, lest corpuses be weighted by outlier projects. 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.

\textbf{B. Implications of regex measurements}

In the preceding sections we applied our measurements to test the validity of the H-SR and H-CL hypotheses. In those tests we considered the relative values of the measures between static and dynamic regexes (H-SR §V) and between static regexes in different languages (H-CL §VI). 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, the 75\textsuperscript{th} percentile and 90\textsuperscript{th} percentile are useful for reasoning about the common case of regexes encountered by regex tools and engines (§III-B).

\textbf{Regex Representation}. Measures of regex representation (pattern, automaton) may be the most relevant for regex visualization and debugging tools. The (25, 75)\textsuperscript{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 [65], match/mismatch aids [66]) should be made to perform well on regexes of these lengths. Similarly, NFA-based regex tools (e.g., railroad diagrams [65]) should accommodate NFAs with between 5 and 30 NFA states, which will cover the (25, 75)\textsuperscript{th} percentile range in every language.

\textbf{Language diversity}. The bottom 90\% of regexes in every language have at most 10 simple paths through their NFA representation; a covering set of 10 inputs is sufficient to enumerate the “equivalence classes” of the NFA. Larson and
Kirk’s basis path-based approach [23] 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, lookahead assertions) in any programming language, Thompson’s algorithm can be applied to almost all regexes in every programming language, addressing 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 Spencer-style algorithms in the rare cases that super-linear features are used. This approach has previously found successful implementation in grep [67]. 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.

Lastly, were regex engine designers to incorporate Thompson’s algorithm, 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 languages the 90th percentile NFA outdegree density is no larger than that of the regexes in Rust (0.38) and Go (0.44). However, in Java the 90th percentile 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).

---

**VIII. Threats to Validity**

**Internal validity.** As noted in §V-A, our approach to generating the static/dynamic corpus was 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.

We considered all unique regexes equally, de-duplicating the regexes by their pattern. 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 or not previous empirical regex findings generalized to dynamic regexes in the same language as well as to regexes written in other languages. We situated our research in the context of open-source software modules. The generalizability of this approach to applications and to closed-source software relies on two assumptions: (1) that regex characteristics will be similar for modules and applications in the same programming language, and (2) that regex characteristics will be similar for open-source and closed-source software in a given programming language. We feel these are reasonable assumptions.

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., in firewalls and intrusion detection systems [28], [68].

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 primary loss was in the Weideman et al. worst-case Spencer analysis, which could measure only about 80% of the regexes. In the rest of the analyses we were able to measure at least 90% of the regexes. Although the missing regexes might have different characteristics, e.g., relying on unusual features, there were not enough of them to significantly impact our findings.

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

**IX. Conclusion**

Previous empirical research on regex characteristics focused on static 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 the characteristics of statically-declared and dynamically-defined regexes are similar, and the characteristics of statically-declared regexes are sometimes similar across programming languages. However, some regex characteristics do not generalize across programming languages, and we encourage future empirical regex researchers to design their studies appropriately. 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.

References


[47] “cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages,” https://github.com/AlDanial/cloc.


