

# Regular Expression Feature Usage in Python

Carl Chapman  
Department of Computer Science  
Iowa State University  
Ames, IA, USA  
carl1978@iastate.edu

Kathryn T. Stolee  
Departments of Computer Science and  
Electrical and Computer Engineering  
Iowa State University  
Ames, IA, USA  
kstolee@iastate.edu

## ABSTRACT

Regular expressions are used frequently in programming languages for form validation, ad-hoc file searches, and simple parsing. Due to the popularity and pervasive use of regular expressions, researchers have created tools to support their creation, validation, and use. Each tool has made design decisions about which regular expression features to support, and these decisions impact the usefulness and power of the tools. Yet, these decisions are often made with little information as there does not exist an empirical study of regular expression feature usage to inform these design decisions.

In this paper, we survey 18 professional developers about the context and frequency of their regular expression usage. Then, we explore regular expression feature usage in Python, focusing on how often features are used and the diversity of regular expressions from syntactic and semantic perspectives. We analyzed nearly 4,000 open source Python projects from GitHub and extracted nearly 14,000 unique regex patterns that were used for analysis. We also map the most common features used in regular expressions to those features supported by four common regex engines from industry and academia, brics, Hampi, Re2, and Rex. Our results indicate that developers frequently use regular expressions in their programming practices, but often those regular expressions do not persist (e.g., when used for command line or file search purposes). For regular expressions found in Python projects, the most commonly used features are also supported by popular research tools and that programmers frequently reinvent the wheel by writing identical or nearly identical regular expressions in different ways. We conclude by discussing the implications for tool designers and outline several directions of future work. **TODO.LAST: Katie: revise**

## 1. INTRODUCTION

Regular expressions (regexes) are an abstraction of keyword search that enables the identification of text using a pattern instead of an exact keyword. Regexes are commonly

used for parsing text, form validation, and text searching within text editors (e.g., emacs), command line tools (e.g., grep, sed) and IDEs (e.g., the search feature in the Eclipse IDE). Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports [18].

Due in part to their common use across programming languages and how susceptible regexes are to error, many researchers and practitioners have developed tools to support more robust creation [18] or to allow visual debugging [6]. To remove the human in the loop, other research has focused on learning regular expressions from text [4, 13]. Beyond supporting regular expression usage, the applications of regular expressions in research include test case generation [2, 9, 10, 20], specification for string constraint solvers [11, 21], and as queries in a data mining framework [7] or on the semantic web [12]. Regexes are also employed in critical missions like mysql injection prevention [23] and network intrusion detection [16], or in more diverse applications like DNA sequencing alignment [3].

These researchers and tool designers must pick what features to include or exclude, which can be a difficult design decision. Supporting advanced features may be more expensive, taking more time and potentially making the project too complex and cumbersome to execute well. A selection of only the simplest of regex features limits the applicability or relevance of that work. Despite extensive research effort in the area of regex support, no research has been done about how regexes are used in practice and what features are essential for the most common use cases.

*The goal of this work is to explore 1) the context in which developers use regular expressions, and 2) the features and diversity of regular expressions found in Python projects.* First, we survey developers about the context of their regex usage, include how often and for what purposes regexes are composed. Second, we measure how often regex features (e.g., kleene star, character classes, and capture groups are all features) appear in regular expressions and used in Python projects. By comparing the features to those supported by four common regex support tools, brics [15], hampi [11], Rex [22], and RE2 [17] and surveying developers about some of the less supported feature, we discuss the potential impact of omitting various features. Third, using a semantic analysis to cluster similar regular expressions, we explore the most common behaviors captured by regexes in Python. Our results indicate that regexes are frequently used in a variety of language and environments, that about half of the scraped Python projects use regexes, and that **TODO.LAST: add**

**short finding from clustering.** The contributions of this work are:

- A survey of 18 professional software developers about their experience with regular expressions,
- An empirical analysis of regex feature usage of nearly 14,000 regular expressions in 3,898 open-source Python projects, mapping of those features to those supported by common regex tools and survey results showing the impact of not supporting various features,
- An approach for measuring semantic similarity of regular expressions and qualitative analysis of the most common semantically similar clusters, and
- A discussion of opportunities for future work in supporting programmers in writing regular expressions.

The rest of the paper is organized as follows. Section 2 motivates this work by discussing research in supporting programmers in the use, creation, and validation of regular expressions. Section 3 presents the research questions, survey design, and study setup for exploring regular expressions in the wild. Results of these explorations are in Section 4 followed by a discussion in Section 5. Threats to validity are in Section 6 and the conclusion is in Section 7.

## 2. RELATED WORK

Regular expressions have been a focus point in a variety of research objectives. From the user perspective, tools have been developed to support more robust creation [18] or to allow visual debugging [6]. Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text [4, 13].

Regarding applications, regular expressions have been used for test case generation [2, 9, 10, 20], and as specifications for string constraint solvers [11, 21]. Regexes are also employed in critical missions like mysql injection prevention [23] and network intrusion detection [16], or in more diverse applications like DNA sequencing alignment [3] or querying RDF data [1, 12].

As a query language, lightweight regular expressions are pervasive in search. For example, some data mining frameworks use regular expressions as queries (e.g., [7]). Efforts have also been made to expedite the processing of regular expressions on large bodies of text [5].

Research tools like Hampi [11], and Rex [22], and commercial tools like brics [15] and RE2 [17], all use regular expressions for various task. Hampi was developed in academia and uses regular expressions as a specification language for a strong constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in several applications, such as test case generation [2, 20]. Brics is an open-source package that creates automata from regular expressions for manipulation and evaluation. RE2 is an open-source tool created by Google to power Code Search with a more efficient regex engine.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterns [14] and bug characterizations [8]. Exploring language feature usage by mining source code has been studied extensively for Smalltalk [?, ?], JavaScript [?], and Java [?,

function	pattern	flags
<code>r1 = re.compile('</code>	<code>(0 -?[1-9][0-9]*)\$'</code>	<code>re.MULTILINE</code>

**Figure 1: Example of one regex utilization**

?, ?, ?], and more specifically, Java generics [?] and Java reflection [?]. To our knowledge, this is the first work to mine and evaluate regular expression usages from existing software repositories. Related to mining work, regular expressions have been used to form queries in mining framework [7], but have not been the focus of the mining activities. Surveys have been used to measure adoption of various programming languages [?, ?], and been combined with software repository analysis [?], but have not focused on regular expressions or on particular regular expression feature usages.

## 3. STUDY

To understand how programmers use regular expressions in Python projects, we scraped 3,898 Python projects from GitHub, and recorded regex usages for analysis. Throughout the rest of this paper, we employ the following terminology:

**Utilization:** A *utilization* occurs whenever a developer uses a regex in a project. We detect utilizations by recording all calls to the `re` module in Python. Within a particular file in a project, a utilization is composed of a function, a pattern, and 0 or more flags. Figure 1 presents an example of one regex utilization, with key components labeled. The function call is `re.compile`, `(0|-?[1-9][0-9]*)$` is the regex string, or pattern, and `re.MULTILINE` is an (optional) flag. This utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag. Thought of another way, a regular expression utilization is one single invocation of the `re` library.

**Pattern:** A *pattern* is extracted from a utilization, as shown in Figure 1. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens. The pattern in Figure 1 will match if it finds a zero at the end of a line, or a (possibly negative) integer at the end of a line (i.e., due to the `-?` sequence denoting zero or one instance of the `-`).

Note that because the vast majority of regex features are shared across most general programming languages (e.g., Java, C, C#, or Ruby), a Python pattern will (almost always) behave the same when used in other languages, whereas a utilization is not universal in the same way (i.e., it may not compile in other languages, even with small modifications to function and flag names). As an example, the `re.MULTILINE` flag, or similar, is present in Python, Java, and C#, but the Python `re.DOTALL` flag is not present in C# though it has an equivalent flag in Java.

In this work, we primarily focus on patterns since they are cross-cutting across languages and are the primary way of specifying the matching behavior for every utilization. Next, we describe the research questions and how the data set was collected and analyzed.

### 3.1 Research Questions

To understand how regular expressions and regular expression features are used in Python projects, we aim to

answer the following research questions:

**RQ1:** In what context do developers use regular expressions?

We designed and deployed a survey about when, why, and how often they use regular expressions. This was completed by 18 professional developers at a small software start-up company.

**RQ2:** How is the `re` module used in Python projects?

We explore invocations of the `re` module in 3,898 Python projects.

**RQ3:** Which regular expression language features are most commonly used in python?

We consider regex language features to be tokens that specify the matching behavior of a regex pattern, for example, the `+` in `ab+`. All studied features are listed and described in Section 3.3 with examples. We then map the feature coverage for four common regex support tools, brics, hampi, RE2 and Rex, and explore survey responses of feature usage for some of the less supported features.

**RQ4:** How semantically similar are regexes?

We measure similarity between pairs of regexes by generating strings that match each and evaluating each other regex against those strings to build a similarity matrix. Then we use clustering to form semantic groupings.

Next, we describe our survey, how the corpus of regex patterns was built, how features were analyzed, and how the clustering was performed.

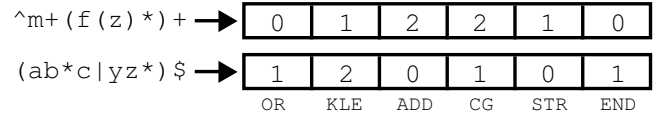
## 3.2 Survey Design

To understand the context of when and how programmers use regular expressions, we designed a survey with 40 questions about regex usage and context. This survey was deployed to 22 developers at Dwolla, a company that provides software for online and mobile payment management. Participation was voluntary and participants were entered in a lottery for a \$50 gift card. The survey was completed by 18 participants (82% response rate) that identified as software developer/maintainers. The questions asked about regex usage frequency, languages, purposes, and the use of various language features, as explored in Section 4.1 and Section 4.3.

## 3.3 Regex Corpus

Using the GitHub API, we scraped 3,898 Python projects. Each project’s commit history was scanned at 20 evenly-spaced commits. If the project had fewer than 20 commits, then all commits were scanned. The most recent commit was always included, and the spacing between all other chosen commits was determined by dividing the remaining number of commits by 19 (rounding as needed). All regex utilizations were obtained, sans duplicates. Within a project, a duplicate utilization was marked when two versions of the same file have the same function, pattern and flags. In the end, we observed and recorded 53,894 non-duplicate regex utilizations in 3,898 projects.

In collecting the set of distinct patterns for analysis, we ignore the 12.7% of utilizations using flags, which can alter regex behavior. An additional 6.5% of utilizations contained patterns that could not be compiled because the pattern was



**Figure 2:** Two patterns parsed into feature vectors

non-static (e.g., used some runtime variable). The remaining 80.8% (43,525) utilizations were collapsed into 13,711 distinct pattern strings. Each of the pattern strings was pre-processed by removing Python quotes (`'\\W'` becomes `\\W`), unescaping escaped characters (`\\W` becomes `\\W`) and parsing the resulting string using an ANTLR-based, open source PCRE parser<sup>1</sup>. This parser was unable to support 0.5% (73) of the patterns due to unsupported unicode characters. Another 0.2% (25) of the patterns used regex features that we chose to exclude because appeared very rarely (e.g., Reference Conditions).

The 13,597 distinct pattern strings that remain were each assigned a weight value equal to the number of distinct projects the pattern appeared in. We refer to this set of weighted, distinct pattern strings as the *corpus*.

## 3.4 Analyzing Features

For each escaped pattern, the PCRE-parser produces a tree of feature tokens, which is converted to a vector by counting the number of each token present in the tree. For a simple example, consider the patterns in Figure 2. The pattern `^m+(f(z)*)+` contains four different types of tokens. It contains the kleene star (KLE), which is specified using the asterisk `*` character, additional repetition (ADD), which is specified using the plus `+` character, capture groups (CG), which are specified using pairs of parenthesis `(...)` characters, and the start anchor (STR), which is specified using the caret `^` character at the beginning of a pattern.

Once all patterns were transformed into vectors, we examined each feature independently for all patterns, tracking the number of patterns and projects that the each feature appears in at least once.

## 3.5 Clustering and Semantic Analysis

Our semantic analysis clusters regular expressions by their behavioral similarity. Consider two unspecified patterns **A** and **B**, a set **mA** of 100 strings that pattern **A** matches, and a set **mB** of 100 strings that pattern **B** matches. If pattern **B** matches 90 of the 100 strings in the set **mA**, then **B** is 90% similar to **A**. If pattern **A** only matches 50 of the strings in **mB**, then **A** is 50% similar to **B**. We use similarity scores to create a similarity matrix as shown in Figure 3. In row **A**, column **B** we see that **B** is 90% similar to **A**. In row **B**, column **A**, we see that **A** is 50% similar to **B**. Each pattern is always 100% similar to itself, by definition.

Once the similarity matrix is built, the values of cells reflected across the diagonal of the matrix are averaged to create a half-matrix of undirected similarity edges, as illustrated in Figure 4. This facilitates clustering using the Markov Clustering (MCL) algorithm<sup>2</sup>. We chose MCL because it offers a fast and tunable way to cluster items by


<sup>1</sup><https://github.com/bkiers/pcpre-parser>

<sup>2</sup><http://micans.org/mcl/>

Pattern A matches 100/100 of A's strings		
Pattern B matches 90/100 of A's strings	A	B
Pattern A matches 50/100 of B's strings	1.0	0.9
Pattern B matches 100/100 of B's strings	0.5	1.0

**Figure 3: A similarity matrix created by counting strings matched**

	A	B	C	D
A	1.0	0.0	0.9	0.0
B	0.2	1.0	0.8	0.7
C	0.6	0.8	1.0	0.2
D	0.0	0.6	0.1	1.0



	A	B	C	D
A	1.0			
B	0.1	1.0		
C	0.75	0.8	1.0	
D	0.0	0.65	0.15	1.0

**Figure 4: Creating a similarity graph from a similarity matrix**

similarity and it is particularly useful when the number of clusters is not known *a priori*.

In the implementation, strings are generated for each pattern using Rex [22]. Rex generates matching strings by representing the regular expression as an automation, and then passing that automation to a constraint solver that generates members for it<sup>3</sup>. If asked to produce more strings than the automation can provide, Rex will instead produce a list of all possible strings. Our goal is to generate 400 strings for each pattern to balance the runtime of the similarity analysis with the precision of the similarity calculations.

We note that MCL can be tuned using many parameters, including inflation and filtering out all but the top-k edges for each node. After exploring the quality of the clusters using various tuning parameter combinations, the best clusters (by inspection) were found using an inflation value of 1.8 and k=83. The end result is clusters of highly semantically similar regular expressions. The top 100 clusters are categorized by inspection into six categories of behavior (see Section 4.4).

## 4. RESULTS

In this section, we present the results of each research question.

### 4.1 RQ1: In what context do developers use regular expressions?

The survey was completed by 18 professional software developers with an average of nine years of programming experience ( $\sigma = 4.28$ ). On average, survey participants report to compose 172 regexes per year ( $\sigma = 250$ ) and compose regexes on average once per month, with 28% composing multiple regexes in a week and an additional 22% composing regexes once per week. That is, 50% of respondents uses regexes at least weekly. Table 1 shows how frequently participants compose regexes using each of several languages and technical environments. Six (33%) of the survey participants report to compose regexes using general purpose

**Table 1: Survey results for number of regexes composed per year by technical environment**

Language/Environment	0	1-5	6-10	11-20	21-50	51+
General (e.g., Java)	1	6	5	3	1	2
Scripting (e.g., Perl)	5	4	3	3	2	1
Query (e.g., SQL)	15	2	0	0	1	0
Command line (e.g., grep)	2	5	3	2	0	6
Text editor (e.g., IntelliJ)	2	5	0	5	1	5

**Table 2: Survey results for regex usage frequencies for various activities, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1**

Activity	Frequency
Locating content within a file or files	4.4
Capturing parts of strings	4.3
Parsing user input	4.0
Counting lines that match a pattern	3.2
Counting substrings that match a pattern	3.2
Parsing generated text	3.0
Filtering collections (lists, tables, etc.)	3.0
Checking for a single character	1.7

programming languages (e.g., Java, C, C#) 1-5 times per year and five (28%) do this 6-10 times per year. Regexes were rarely used in query languages like SQL, but for command line usage in tools such as grep, 6 (33%) participants use regexes 51+ times per year.

Table 2 shows how frequently, on average, the participants use regexes for various activities. Participants answered questions using a 6-point likert scale including very frequently, frequently, occasionally, rarely, very rarely, and never. Assigning values from 1 to 6, where 6 is the most frequent, the responses were averaged across participants. Among the most common usages are capturing parts of a string and locating content within a file, with both occurring somewhere between occasionally and frequently.

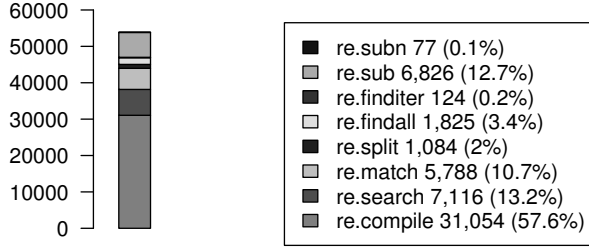
Using a similar 7-point likert scale that includes ‘always’ as a seventh point, developers indicated that they test their regexes with the same frequency as they test their code (average response was 5.2, which is between frequently and very frequently). Half of the 18 developers indicate that they use external tools to test their regexes, and the other half indicated that they only use tests that they write themselves. Of the nine developers using tools, six mentioned some online composition aide such as [regex101.com](http://regex101.com) where a regex and input string are entered, and the input string is highlighted according to what is matched.

When asked an open ended question about pain points encountered with regular expressions, we observed three main categories. The most common, “hard to compose,” was represented in 61% (11) responses. Next, 39% (7) developers responded that regexes are “hard to read” and 17% (3) indicated difficulties with “inconsistency across implementations,” which manifest when using regexes in multiple languages. These responses do not sum to 18 as three developers provided overlapping answers.

<sup>3</sup><http://research.microsoft.com/en-us/projects/rex/>

**Table 3: How Saturated are Projects with Utilizations? (RQ1)**

source	Q1	Avg	Med	Q3	Max
utilizations per project	2	32	5	19	1,427
files per project	2	53	6	21	5,963
utilizing files per project	1	11	2	6	541
utilizations per file	1	2	1	3	207



**Figure 5: How often are re functions used? (RQ1)**

#### Summary - RQ1.

Overall, regexes are used frequently with common usages including locating content within a file, capturing parts of strings, and parsing user input. The fact that all the surveyed developers compose regexes, and half of the developers use tools to test their regexes indicates the importance of tool development for regex. Developers complain about regex being hard to read and hard to compose, and most of the tools that they indicate using are focused on composition.

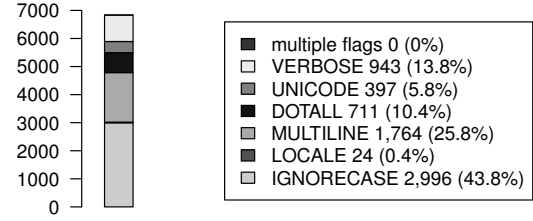
## 4.2 RQ2: How is the re module used?

We explore regex utilizations and flags used in the scraped Python projects. Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one regex utilization. To illustrate how saturated projects are with regexes, we measure utilizations per project, files scanned per project, files contained utilizations, and utilizations per file, as shown in Table 3.

The average project contained 32 utilizations, and the maximum number of utilizations was 1,427. The project with the most utilizations is a C# project<sup>4</sup> that maintains a collection of source code for 20 Python libraries, including larger libraries like `pip`, `celery` and `ipython`. These larger Python libraries contain many utilizations. From Table 3, we also see that each project had an average of 11 files containing any utilization, and each of these files had an average of 2 utilizations.

The number of projects that use each of the `re` functions is shown in Figure 5. The y-axis denotes the total utilizations, with a maximum of 53,894. The `re.compile` function encompasses 57.6% of all utilizations, presumably because each usage of those functions can accept a regex object compiled using `re.compile` as an argument.

When considering flag use, we excluded the default flag, which is built into the `re` module, and present internally



**Figure 6: Which behavioral flags are used? (RQ1)**

whenever no flag is used. Of all utilizations, 87.3% had no flag, or explicitly specified the default flag. The debug flag, which causes the `re` regex engine to display extra information about its parsing process, was never observed. Figure 6 presents the number of projects in which each flag appears. Ignorecase (43.8%) and multiline (25.8%) were the most frequently used. Although multiple flags can be combined using a bitwise or, this was never observed.

#### Summary - RQ2.

Only about half of the projects sampled contained any utilizations. Most utilizations used the `re.compile` function to compile a regex object before actually using the regex to find a match. Most utilizations did not use a flag to modify matching behavior.

## 4.3 RQ3: Which regular expression language features are most commonly used in Python?

To measure feature usage, we count the number of usages of each feature per project and as a percent of all distinct regular expression patterns in the corpus.

### 4.3.1 Feature Usage

Table 4 displays feature usage from the corpus and relates it to four major regex related projects. Only features appearing in at least 10 projects are listed. The first column, *rank*, lists the rank of a feature (relative to other features) in terms of the number of projects in which it appears. The next column, *code*, gives a succinct reference string for the feature, and is followed by a *description* column that provides a brief comment on what the feature does. The *example* column provides a short example of how the feature can be used. The next four columns, (i.e., *brics*, *hampi*, *Rex*, and *RE2*), map to the four major research projects chosen for our investigation (see Section 4.3.2). We indicate that a project supports a feature with the ‘●’ symbol, and indicate that a project does not support the feature with the ‘○’ symbol. The final four columns contain two pairs of usage statistics. The first pair contains the number and percent of *patterns* that a feature appears in, out of the 13,912 patterns that make up the corpus. The second pair of columns contain the number and percent of *projects* that a feature appears in out of the 1645 projects scanned that contain at least one utilization.

One notable omission from Table 4 is the literal feature, which is used to specify matching any specific character. An example pattern that contains only one literal token is the pattern ‘a’. This pattern only matches the lowercase letter ‘a’. The literal feature was found in 97.7% of patterns. We consider the literal feature to be ubiquitous in all patterns,

<sup>4</sup><https://github.com/Ouroboros/Arianrhod>

Table 4: How frequently do features appear in projects? (RQ2)

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nProjects	% projects
1	ADD	one-or-more repetition	<code>z+</code>	●	●	●	●	6,003	44.1	1,204	73.2
2	CG	a capture group	<code>(caught)</code>	●	●	●	●	7,130	52.4	1,194	72.6
3	KLE	zero-or-more repetition	<code>.*</code>	●	●	●	●	6,017	44.3	1,099	66.8
4	CCC	custom character class	<code>[aeiou]</code>	●	●	●	●	4,468	32.9	1,026	62.4
5	ANY	any non-newline char	<code>.</code>	●	●	●	●	4,657	34.3	1,005	61.1
6	RNG	chars within a range	<code>[a-z]</code>	●	●	●	●	2,631	19.3	848	51.6
7	STR	start-of-line	<code>^</code>	○	●	●	●	3,563	26.2	846	51.4
8	END	end-of-line	<code>\$</code>	○	●	●	●	3,169	23.3	827	50.3
9	NCCC	negated CCC	<code>[~qwx]</code>	●	●	●	●	1,935	14.2	776	47.2
10	WSP	<code>\t \n \r \v \f</code> or space	<code>\s</code>	○	●	●	●	2,846	20.9	762	46.3
11	OR	logical or	<code>a b</code>	●	●	●	●	2,102	15.5	708	43
12	DEC	any of: 0123456789	<code>\d</code>	○	●	●	●	2,297	16.9	692	42.1
13	WRD	<code>[a-zA-Z0-9_]</code>	<code>\w</code>	○	●	●	●	1,430	10.5	650	39.5
14	QST	zero-or-one repetition	<code>z?</code>	●	●	●	●	1,871	13.8	645	39.2
15	LZY	as few reps as possible	<code>z+?</code>	○	●	○	●	1,300	9.6	605	36.8
16	NCG	group without capturing	<code>a(?:b)c</code>	○	●	○	●	791	5.8	404	24.6
17	PNG	named capture group	<code>(?P&lt;name&gt;x)</code>	○	●	○	●	915	6.7	354	21.5
18	SNG	exactly n repetition	<code>z{8}</code>	●	●	●	●	581	4.3	340	20.7
19	NWSP	any non-whitespace	<code>\S</code>	○	●	●	●	484	3.6	270	16.4
20	DBB	$n \leq x \leq m$ repetition	<code>z{3,8}</code>	●	●	●	●	367	2.7	238	14.5
21	NLKA	sequence doesn't follow	<code>a(?!yz)</code>	○	○	○	○	131	1	183	11.1
22	WNW	word/non-word boundary	<code>\b</code>	○	○	○	●	248	1.8	166	10.1
23	NWRD	non-word chars	<code>\W</code>	○	●	●	●	94	0.7	165	10
24	LWB	at least n repetition	<code>z{15,}</code>	●	●	●	●	91	0.7	158	9.6
25	LKA	matching sequence follows	<code>a(=?bc)</code>	○	○	○	○	112	0.8	158	9.6
26	OPT	options wrapper	<code>(?i)CasE</code>	○	●	○	●	231	1.7	154	9.4
27	NLKB	sequence doesn't precede	<code>(?&lt;!x)yz</code>	○	○	○	○	94	0.7	137	8.3
28	LKB	matching sequence precedes	<code>(?&lt;=a)bc</code>	○	○	○	○	80	0.6	120	7.3
29	ENDZ	absolute end of string	<code>\Z</code>	○	○	○	●	89	0.7	90	5.5
30	BKR	match the $i^{th}$ CG	<code>\1</code>	○	○	○	○	60	0.4	84	5.1
31	NDEC	any non-decimal	<code>\D</code>	○	●	●	●	36	0.3	58	3.5
32	BKRN	references PNG	<code>\g&lt;name&gt;</code>	○	●	○	○	17	0.1	28	1.7
33	VWSP	matches U+000B	<code>\v</code>	○	○	●	●	13	0.1	15	0.9
34	NWNW	negated WNW	<code>\B</code>	○	○	○	●	4	0	11	0.7

and necessary for any regex related tool to support, and so exclude it from Table 4 and the rest of the feature analysis.

The eight most commonly used features, ADD, CG, KLE, CCC, ANY, RNG, STR and END, appear in over half the projects. CG is more commonly used in patterns than the highest ranked feature (ADD) by a wide margin (over 8%), even though they appear in similar numbers of projects.

#### 4.3.2 Feature Support in Regex Tools

While there are many regex tools available, in this work, we focus on the features support for these four tools, brics, hampi, Rex and RE2, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and

across applications. Further, as we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

To create the tool mappings, we consulted documentation for each of the selected regular expression engines. For brics, we collected the set of supported features using the formal grammar<sup>5</sup>. For hampi, we manually inspected the set of regexes included in the `lib/regex-hampi/sampleRegex` file within the hampi repository<sup>6</sup> (this may have been an over-

<sup>5</sup><http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html>

<sup>6</sup><https://code.google.com/p/hampi/downloads/list>

**Table 5: Survey results for preferences between custom character and default character classes.**

Preference	Frequency
use only CCC	1
use CCC more than default	5
use both equally	2
use default more than CCC	10
use only default	2

estimation, as this included more features than specified by the formal grammar<sup>7</sup>). For RE2, we used the supported feature documentation<sup>8</sup>. For Rex, we collected the feature set empirically because we tried to parse all patterns with Rex for the semantic analysis (Section 4.4), and Rex provides comprehensive error feedback for unsupported features.

Of the four projects selected for this analysis, RE2 supports the most studied features (28 features) followed by hampi (25 features), Rex (21 features), and brics (12 features). All projects support the 8 most commonly used features except brics, which does not support STR or END. No projects support the four look-around features LKA, NLKA, LKB and NLKB. RE2 and hampi support the LZY, NCG, PNG and OPT features, whereas brics and Rex do not.

### 4.3.3 Survey Results for Feature Usage

The pattern language for Python, which is used to compose regexes, supports default character classes like the ANY or dot character class: `.` meaning, ‘any character except newline’ (a full list of features and examples is in the first four columns of Table 4). It also supports three other default character classes: `\d`, `\w`, `\s` (and their negations). All of these default character classes can be simulated using the custom character class (CCC) feature, which can create semantically equivalent regexes. For example the decimal character class: `\d` is equivalent to a CCC containing all 10 digits: `\d`  $\equiv$  `[0123456789]`  $\equiv$  `[0-9]`. Other default character classes such as the word character class: `\w` may not be as intuitive to encode in a CCC: `[a-zA-Z0-9_]`.

Survey participants were asked if they use only CCC, use CCC more than default, use both equally, use default more than CCC or use only default. Results for this question are shown in Table 5, with 67% (12) indicating that they use default the most. Participants were also asked to explain their preferences. Participants who favored CCC mostly said something equivalent to “it is more explicit,” whereas the participants who favored default character classes said, “it is less verbose” and “I like using built-in code.”

To further explore how frequently participants use various regex features, participants were asked five questions (on a 6-point likert scale) about how frequently they use specific related groups of features, chosen based on the tool feature support explored in Section 4.3.2. Results are shown in Table 6, indicating that lazy repetition and look-ahead features are rarely used and capture groups and endpoint anchors are occasionally to frequently used.

**Table 6: Survey results for regex usage frequencies, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1**

Group	Code	Frequency
endpoint anchors	(STR, END)	4.4
capture groups	(CG)	4.2
word boundaries	(WNW)	3.5
lazy repetition	(LZY)	2.9
(neg) look-ahead/behind	(LKA, NLKA, LKB, NLKB)	2.5

### Summary - RQ3.

We found that the eight most common features are found on over 50% of the projects. We also identify brics as the project supporting the fewest features and RE2 as the project supporting the most features, and identify groups of features supported or not supported by the four regex projects. Some of the less supported features, such as endpoint anchors and capture groups, are used by developers occasionally to frequently.

From Table 4 we know that STR and END features are present in over half of the scanned projects containing utilizations. In our survey, over half (56%) of the respondents answers that they use endpoint anchors frequently or very frequently, and none of them claimed to never use them. The brics library does not support this feature, which is a missed opportunity for developers who could otherwise have used brics to model their regexes that use STR and END.

The LZY feature is present in over 36% of scanned projects with utilizations (Table 4), and yet was not supported by two of the four major regex projects we explored, brics and RE2. In our developer survey, 11% (2) of participants use this feature frequently and 6 (33%) use it occasionally, showing a modest impact on potential users.

When survey participants were asked if they prefer to always use numbered (BKR) or named (BKRn) back references, 66% (12) of survey participants said that they always use BKR, and the remaining 33% (6) said “it depends.” Not one participant preferred named capture groups. BKR is present in 5% of scanned projects, while BKRn is present in only 1.7%, which corroborates our findings that numbered are generally preferred over named capture groups.

## 4.4 RQ4: How semantically similar are regexes?

Since Rex does not support all the features present in the corpus, and because we discarded patterns found in only one project, we only generated sets of matching strings for 2,871 (21%) of the 13,597 patterns in the corpus. The impact is that 923 projects were excluded from the data set for the similarity analysis. Omitted features are indicated in Table 4, as described in Section 4.3. The generated strings for each pattern are used to measure the pairwise similarity for all patterns and construct the similarity matrix.

When tool designers are considering what features to include, data about usage in practice is valuable. Semantic similarity clustering helps to discern these behaviors by looking beyond the structural details of specific patterns and seeing trends in actual matching behavior. We are also able to find out what features are being used in these behavioral

<sup>7</sup><http://people.csail.mit.edu/akiezun/hampi/Grammar.html>

<sup>8</sup><https://re2.googlecode.com/hg/doc/syntax.html>



**Table 7: An example cluster (RQ3)**

index	pattern	nProjects	index	pattern	nProjects
1	'\(.*)'	54	5	'(.*)\((.*)\)'	3
2	'\((.*)\)'	30	6	'.*\((.*)\).*	2
3	'\([~]*\)'	16	7	'(.*)\(.*)'	1
4	'\([~()]*\)'	0			

trends so that we can make assertions about why certain features are important.

From 2,871 distinct patterns, the MCL clustering technique identified 186 clusters with 2 or more patterns, and 2,042 clusters of size 1. Recall that only pairs of patterns with a similarity level of 0.75 were included in the matrix passed to MCL. The average size of clusters larger than size one was 4.5. Each pattern belongs to exactly one cluster.

Table 7 provides an example of a behavioral cluster representing 13 patterns with at least one pattern from this cluster present in 100 different projects. At first glance this cluster may seem to revolve around the '\s\*' parts of these patterns, but actually this cluster was formed because each of these patterns has a comma literal, and other details did not interfere with matching the Rex-generated strings with commas in them.

The smallest pattern in Table 7 is the single comma literal ',', at index 1. This smallest pattern gives the a good idea of what all the patterns within it have in common. A shorter pattern will tend to have less extraneous behavior because it is specifying less behavior. And yet in order for the smallest pattern to be clustered with other patterns, it had to match most of the strings created by Rex from another pattern within the cluster, and so we assume that *the smallest pattern is a good representation of the cluster*.

For the rest of this paper, a cluster will be represented by one of the shortest patterns it contains, followed by the number of projects a member of the cluster appears in, so the cluster in Table 7 will be represented as ',', (100).

We manually mapped the top 100 clusters into each of 6 behavioral categories (determined by inspection), omitting 40 clusters that did not fit into any of these categories. ...including 29 clusters composed of long strings (example: `set_fabric_sense_len\()\()` which were in the top 100 because 28 projects that were scanned were forked linux kernels.

Next, we define the six categories and provide examples from the relevant clusters.

#### 4.4.1 Single Literal Characters

This category contains 19 clusters representing **TODO.NOW: X** projects. Each of the clusters center around a single literal character. For example, three of the top clusters in this category include: '\'(110)', ',', (100), and ':'(91). This is in contrast to the survey in Section ?? in which participants reported to very rarely or never use regexes to check for a single character (Table 2).

#### 4.4.2 Default Character Classes

This category contains 12 clusters. Each of the clusters revolves around the use of a default character class. For exam-

ple, three of the top clusters in this category are: '\s'(277), '\W'(208), and '\d'(193). This corroborates our survey results to the question, *Do you prefer to use custom character classes or default character classes more often?*, in which 56% (10) of the participants indicated they use the default classes more than custom.

#### 4.4.3 User-Defined Character Classes

This category contains 10 clusters. Each of the clusters center around user defined character classes. For example, three of the top clusters in this category are: '[a-zA-Z]'(138), '[~]'(122)!, and '[&]'(50). This further supports our survey results in which 33% (6) participants indicated they use the custom classes more than default. The remaining two participants use both equally.

#### 4.4.4 Matching Whole Strings

This category contains 8 clusters. Each of the clusters begins with the STR anchor and ends with the END anchor, requiring the entire input string to match the pattern. For example, three of the top clusters in this category include: '^d+\$'(78), '^w+\$'(74), and '^s+\$'(59).

#### 4.4.5 Parsing Angle Bracket Contents

This category contains 5 clusters. Each of the clusters contains a pair of angle brackets that contain a repeating character class. It appears that these clusters are being used to recognize or capture the contents of the angle brackets. For example, three of the top clusters in this category include: '<.+>'(63), '<!\s+([<>]\*)>'(35), and '<([<>]\*)/>'(35).

#### 4.4.6 Capturing Variable Assignments

This category contains 4 clusters. Each of the clusters contain an equals symbol and some pattern on either side of it, which appears to be a variable on the left of the equals sign and a value on the right. This type of cluster is very likely used to capture the value of the variable assignment when parsing source code. For example, three of the top clusters in this category are: '\nmd5\_data = {n([~]+)}'(69), '.\*rlen=([0-9]+)'(57), and 'coding[:]=\s\*([-w.]+)'(48).

### Summary - RQ4.

We used the behavior of individual patterns to form clusters, and identified six main categories that clusters belonged to. Overall, we see that many clusters are defined by the presence of particular tokens, such as the comma for the cluster in Table 7. These six categories define what users are doing with regexes at a high level: using default character classes, defining their own character classes, matching single characters, parsing variable assignments, parsing the contents of brackets, or matching whole lines. The higher frequency of default character class matching than custom character class matching corroborates our developer survey responses. One of the six common cluster categories, *capturing variable assignments*, has a very specific purpose of parsing source code files. This shows a very specific and common use of regular expressions in practice.

## 5. DISCUSSION

In this section, we discuss the implications of these empirical findings on tool designers and users of regex tools and opportunities for future work.



## 5.1 Implications For Tool Designers

The results in this work have several implications for tool designers who want to effectively support developers who use regular expressions.

### 5.1.1 Capturing Specific Content

The survey results from Section 4.1 indicate that capturing parts of strings is among the most frequent activities for which developers use regexes. The capture group (CG) feature is the most frequently used feature in terms of patterns (Table 4). As mentioned in Section 4.4, capturing values assigned to variables when parsing source code was one of the main categories of clusters observed. **TODO.MID: update?** Thus, we observe that the ability to capture some part of a match provides a powerful tool to programmers. The CG feature has two functions: 1) it allows logical grouping as would be expected by parenthesis, and 2) it allows retrieval of information in one logical grouping. Any non-trivial tool or research that hopes to be applicable to regex use in practice must treat the CG feature as especially important, and must support some way to reason about what information is retrieved by capture groups.

### 5.1.2 File Parsing

Text files containing one unit of information per line are common in a wide variety of applications (for example log and csv files). Out of the 13,912 patterns in the corpus, 3,444 (24%) contained ANY followed by KLE (i.e., ‘.\*’), often at the end of the pattern. One reasonable explanation for this tendency to put ‘.\*’ at the end of a pattern is that users want to disregard all matches after the first match on a single line in order to count how many distinct lines the match occurs on. Survey participants indicated an average frequency of “Counting lines that match a pattern” and “Counting substrings that match a pattern” at 3.2 or Rarely/Occasionally.

Delimiters that separate items on one line like ‘,’ are also quite common. Although survey participants indicated an average frequency of 1.7 (very rarely or never) for “checking for a single character,” we found 19 clusters whose essential behavior was to search for one or two characters. **TODO.MID: update!** Yet, the top ranked activity for developers is “Locating content within a file or files,” and usually this content is located using some small set of characters that the user knows will flag that content. Looking closely at that category of cluster, some of the characters being searched for were -, # and : - all common delimiters in different scenarios. This emphasizes the need for regex tools to facilitate easy file parsing.

## 5.2 Opportunities For Future Work

Based on our findings, there are many opportunities for future work.

### 5.2.1 Context-Specific Regex Support

In some environments, such as command line or text editor, regexes are used extensively (Section 4.1), but these regular expressions do not persist. Thus, using a repository analysis for feature usage only illustrates part of how regexes are used in practice. Exploring how the feature usage differs between environments would help inform tool developers about how to best support regex usage in context, and is left for future work.

### 5.2.2 Refactoring Regexes

The survey showed that users want readability and find the lack of readable regexes to be a major pain point. This provides an opportunity to introduce refactoring transformations to enhance readability. As one opportunity, certain character classes that are logically equivalent can be expressed differently, for example, `\d`  $\equiv$  `[0123456789]`  $\equiv$  `[0-9]`. While `\d` is more succinct, `[0-9]` may be easier to read, so a refactoring for *default to custom character classes* could be introduced. Human studies are needed to evaluate the readability of various regex features in order to define and support appropriate regex refactorings for readability.

Another avenue of refactoring could be for performance. Various implementations of regex libraries may perform more efficiently with some features than others. An evaluation of regex feature implementation speeds would facilitate semantic transformations based on performance, similar to performance refactorings for LabVIEW [?, ?].

### 5.2.3 Developer Awareness of Best Practices

**TODO.MID: update based on clusters** One category of five clusters in the top 100 contained regex patterns to parse the contents of angle brackets. As the contents of angle brackets is usually unconstrained, regexes are a poor replacement for XML or HTML parsers. More research is needed into how regex users discover best practices and how aware they are of how regexes should and should not be used.

### 5.2.4 Library Support for Developers

Within standard programming languages, regular expressions libraries are very common, yet there are differences between languages in the features that they support. For example, Java supports possessive quantifiers like ‘`ab*+c`’ (here the ‘+’ is modifying the ‘\*’ to make it possessive) whereas Python does not. Such differences among programming language implementations was identified as a pain point for using regular expressions by 17% of the survey participants. **TODO.MID: ”so what?”**

### 5.2.5 Automated Regex Repair

Regular expression errors are common and have produced thousands of bug reports [18]. This provides an opportunity to introduce automated repair techniques to fix. Recent approaches to automated program repair rely on mutation operators to make small changes to source code and then re-run the test suite (e.g., [?, ?]). In regular expressions, it is likely that the broken regex is close, semantically, to the desired regex. Syntax changes can lead to big changes in semantics, so we hypothesize that using the semantic clusters identified in Section 4.4 to identify potential repair candidates would efficiently and effectively converge on a repair candidate.

### 5.2.6 A Modern WRD Character Class

**TODO.MID: update** One unexpected result of our clustering is that, behaviorally speaking, the negation of the word class NWRD was used in 208 projects, while the word class itself was used in only 114 projects. After inspecting several projects using the patterns found in this behavioral cluster, we concluded that most users are trying to sanitize arbitrary strings that must conform to a system character set requirement, such as requirements for filenames. For ex-

ample, a user might replace all NWRD matching characters with the ‘\_’ to guarantee that an arbitrary string can be used as a filename. We also considered the largest cluster using custom character classes (‘[<sup>^</sup> -~]’(122)) and concluded that users are constructing a more permissive version of the NWRD character class, to allow more non-letter, non-digit characters than just the ‘\_’ in their sanitized strings. More research is needed to determine if a more modern WRD class could be useful, and if so, what characters set is preferred.

## 6. THREATS TO VALIDITY

The threats to validity of this work stem primarily from the reliability of measures, instrumentation, selection bias, hypothesis guessing, and the representativeness of participants and projects.

### 6.1 Conclusion

**Reliability of Measures:** The validity of our survey results is dependent on the clarity of the questions. The authors went through several iterations of the survey and included examples for all the regex feature descriptions to improve understandability.

The similarity measure between regexes used in the cluster algorithm is computed empirically rather than analytically. The larger the number of strings used to generate the similarity matrix, the higher the similarity measure between two truly semantically similar regexes. Our experiments used 400 strings to balance performance and precision, but a higher number could lead to more cohesive clusters. Further, in our clustering algorithm, we require a similarity level of 0.75 so that we do not over-estimate the similarity between regular expressions.

### 6.2 Internal

**Instrumentation:** Regular expression patterns were clustered using strings generated by the Rex tool. This introduces two threats to validity. We assume that the strings generated by Rex are reasonably diverse to help characterize the regular expression behavior. To mitigate this threat, Rex generated 400 strings per regular expression and we inspected strings randomly to ensure diversity.

Implementation errors are a risk for research involving repository analysis. To combat this, we have developed extensive test suites to improve our confidence in the results.

**Selection:** We mined only 3,898 Python projects from GitHub, which is small in comparison with the over 100,000 available Python projects. The projects were mined using the GitHub API which sorts the projects by creation date. By using the API, the goal was to reduce any sampling bias introduced by the researchers.

We also did not scrape all commits in every project for regular expression utilizations, rather, we grabbed each project every 20 commits. It is possible that in between the scanned commits, a regular expression utilization was added and then removed, leading to fewer utilizations in our final data set.

### 6.3 Construct

**Hypothesis guessing:** The lead author was an employee of Dwolla when the survey was run, so participant behavior may have been affected by their relationship with the author. To combat this, the beginning of the survey stated, “As this

is for research, please answer as accurately as possible, not guessing what is the desired answer and providing that.”

## 6.4 External

**Interaction of Selection and Treatment:** Our survey participants were software developers from a small startup company. However, the participants may not be representative of all developers. Given that the average participant has nine years of development experience, their responses likely pull from a variety of experiences with regular expression usage.

**Interaction of Setting and Treatment** We only explore regular expressions in Python projects so these results may not generalize to other languages, yet the surveyed participants report to use regexes in a variety of settings and languages. Future work will replicate this study in other languages and compare the results.

## 7. CONCLUSION

Regular expressions are used frequently in programming projects. In our analysis of nearly 4,000 Python projects scraped from GitHub, we observed that over 42% contained a regular expression. The most commonly found regular expressions deal with matching whitespace or digits. In analyzing the features used in regular expressions, we find that the most common is the + token. When clustering regexes based on semantic similarity, we observe that programmers frequently create identical regular expressions using slightly different patterns. After mapping the features supported by four popular regex tools from academia and industry, we observe that most of the most popular features are also supported. However, those unsupported can have an impact on the tool users.

**TODO.MID: add: mining regexes from repos represents just a fraction of all regexes composed by developers. A deeper study into programmer behavior regarding regex usage in all tools, not just those that appear in repos, is needed, and left for future work**

## Acknowledgment

This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, and the Harpole-Pentair endowment at Iowa State University.

## 8. REFERENCES

- [1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending sparql with regular expression patterns (for querying rdf). *Web Semant.*, 7(2):57–73, Apr. 2009.
- [2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.
- [3] A. Arslan. Multiple sequence alignment containing a sequence of regular expressions. In *Computational Intelligence in Bioinformatics and Computational Biology, 2005. CIBCB ’05. Proceedings of the 2005 IEEE Symposium on*, pages 1–7, Nov 2005.
- [4] R. Babbar and N. Singh. Clustering based approach to learning regular expressions over large alphabet for

- noisy unstructured text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data*, AND '10, pages 43–50, New York, NY, USA, 2010. ACM.
- [5] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, Nov. 1996.
  - [6] 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, pages 504–507, New York, NY, USA, 2014. ACM.
  - [7] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.
  - [8] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91, New York, NY, USA, 2014. ACM.
  - [9] S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *Int. J. Softw. Tools Technol. Transf.*, 16(6):727–751, Nov. 2014.
  - [10] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.
  - [11] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, Feb. 2013.
  - [12] J. Lee, M.-D. Pham, J. Lee, W.-S. Han, H. Cho, H. Yu, and J.-H. Lee. Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM Fourth International Workshop on Data and Text Mining in Biomedical Informatics*, DTMBIO '10, pages 23–30, New York, NY, USA, 2010. ACM.
  - [13] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.
  - [14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshypanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.
  - [15] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. <http://www.brics.dk/automaton/>.
  - [16] The Bro Network Security Monitor. <https://www.bro.org/>, May 2015.
  - [17] RE2. <https://github.com/google/re2>, May 2015.
  - [18] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.
  - [19] K. T. Stolee and S. Elbaum. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Trans. Soft. Eng.*, 39(12):1654–1679, 2013.
  - [20] N. Tillmann, J. de Halleux, and T. Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, New York, NY, USA, 2014. ACM.
  - [21] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.
  - [22] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.
  - [23] A. S. Yeole and B. B. Meshram. Analysis of different technique for detection of sql injection. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, pages 963–966, New York, NY, USA, 2011. ACM.