The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale

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

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

Francisco Servant
Virginia Tech, USA
fservant@vt.edu

Dongyoon Lee
Virginia Tech, USA
dongyoon@vt.edu

ABSTRACT

Regular expressions (regexes) are a popular and powerful means of automatically manipulating text. Regexes are also an understudied denial of service vector (ReDoS). If a regex has super-linear worst-case complexity, an attacker may be able to trigger this complexity, exhausting the victim's CPU resources and causing denial of service. Existing research has shown how to detect these super-linear regexes, and practitioners have identified super-linear regex anti-pattern heuristics that may lead to such complexity.

In this paper, we empirically study three major aspects of ReDoS that have hitherto been unexplored: the incidence of super-linear regexes in practice, how they can be prevented, and how they can be repaired. In the ecosystems of two of the most popular programming languages — JavaScript and Python – we detected thousands of super-linear regexes affecting over 10,000 modules across diverse application domains. We also found that the conventional wisdom for super-linear regex anti-patterns has few false negatives but many false positives; these anti-patterns appear to be necessary, but not sufficient, signals of super-linear behavior. Finally, we found that when faced with a super-linear regex, developers favor revising it over truncating input or developing a custom parser, regardless of whether they had been shown examples of all three fix strategies. These findings motivate further research into ReDoS, since many modules are vulnerable to it and existing mechanisms to avoid it are insufficient. We believe that ReDoS vulnerabilities are a larger threat in practice than might have been guessed.

"Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems."
–Jamie Zawinski

CCS CONCEPTS

• **Software and its engineering** → **Empirical software validation**; *Software libraries and repositories*; • **Security and privacy** → *Denial-of-service attacks*;

KEYWORDS

Regular expressions, ReDoS, catastrophic backtracking, empirical software engineering, mining software repositories

ACM Reference Format:
James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236027

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236027

1 INTRODUCTION

Regular expressions (regexes) are a popular and powerful means of automatically manipulating text. They have been applied in a large variety of application domains, *e.g.,* data validation, data scraping, and syntax highlighting [54], and are a standard topic in practical programming texts [16, 28, 39].

Unfortunately, in most popular programming languages, regexes also form an understudied denial of service vector: regular expression denial of service (ReDoS) [23, 24]. For example, in 2016 ReDoS led to an outage at StackOverflow [27] and rendered vulnerable any websites built with the popular Express.js framework [17]. ReDoS attacks are possible because many popular languages, including JavaScript-V8 (Node.js), Python, Java, C++-11, *C#*-Mono, PHP, Perl, and Ruby, rely on a regex engine with worst-case *super-linear behavior* (SL behavior). When these regex engines evaluate *super-linear regexes* (SL regexes) against *malign input*, the evaluation takes polynomial or exponential time in the length of the input, and the high-complexity evaluation overloads the server and denies service to other clients.

ReDoS attacks were first proposed by Crosby in 2003 [23]. In the 15 years since then we have seen advances in detecting SL regexes [34, 37, 38, 47, 51, 55] as well as the introduction of language-level [3] and engine-level [11, 14] defenses against SL behavior. Others have provided conventional wisdom about SL regex anti-patterns, *i.e.,* heuristics to identify forms of a regex that are expected to be particularly risky [30, 32, 33]. Missing, however, is an empirical assessment of the incidence of SL regexes in practice. Simply put, we do not know whether ReDoS is a parlour trick or a security vulnerability common enough to merit further research.

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

In this paper, we perform the first large-scale empirical study to understand the extent of SL regexes in practice as well as the mechanisms that could be used to identify and repair them. We analyze the ecosystems of two of the most popular programming languages to understand the incidence of SL regexes. Our study covers the Node.js (JavaScript) and Python core libraries, as well as 448,402 (over 50%) of the modules in the *npm* [12] and *pypi* [13] module registries. We also study reports of ReDoS in these registries to understand the fixes that developers provide for SL regexes.

We found that SL regexes are rather common: they appear in the core Node.js and Python libraries as well as in thousands of modules in the npm and pypi registries, including popular modules with millions of downloads per month. We found over 4,000 unique SL regexes across npm and pypi, covering a wide range of application domains. Furthermore, nearly 300 of these regexes are high-risk because they have exponential complexity. We disclosed to maintainers the presence of SL regexes in 284 modules, of which 48 have been repaired so far. We found that developers repair SL regexes using one of three techniques: trimming the input, revising the regex, or replacing it with alternate logic. Among these techniques, revising the regex was the most common, regardless of whether developers were previously aware of the others.

This paper provides the following contributions:

- We provide an empirical understanding of the extent (§4.1), seriousness (§4.2), and distribution across application domains (§4.3) of the incidence of SL regexes in two prominent software ecosystems — Node.js and Python.
- We provide an empirical understanding of the effectiveness of the conventional wisdom regarding SL regex anti-patterns (§5.1).
- We provide an empirical understanding of the strategies that developers use to fix SL regexes (§6).

2 BACKGROUND

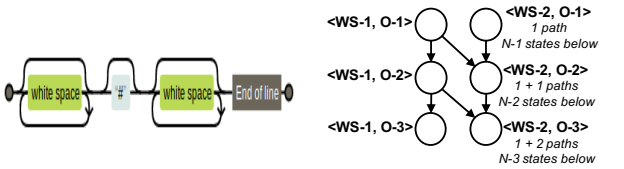
In this section, we review the behavior of SL regexes, their application to ReDoS, and existing techniques to address them.

2.1 Super-Linear (SL) Regex Engines

Regex engines accept a regex describing a language, and an input to be tested for membership in this language. At the core of most regex engines is a backtracking-based search algorithm like the one described by Spencer [44]. A *backtracking regex engine* constructs a non-deterministic finite automaton (NFA) from the regex and then simulates the NFA on the input [42]. The state of the NFA is represented by a 2-tuple: its current vertex in the graph, and its current offset in the input. To simulate non-determinism, whenever the engine makes a choice it pushes the current NFA state onto a stack of decision points. If a mismatch occurs, the engine backtracks to a previous decision point to try another choice. This process continues until the engine finds a match or exhausts the set of decision points. Since each step of an NFA evaluation takes constant time, the complexity of an evaluation corresponds to the number of states that are explored.

A backtracking regex engine may have large worst-case complexity if it does not take care to avoid redundant state exploration[1].

[1]SL features like backreferences are another source of SL worst-case complexity. SL behavior as a result of these features is out of the scope of this work.



(a) Railroad diagram [4]. **(b) NFA execution diagram (excerpt).**

Figure 1: Diagrams for /\s*#?\s*$/ on malign input.

To illustrate, suppose that two choices from a decision point reach the same NFA state. Since from this state a fixed set of states will be explored, this mutual state need be explored only once [31]. However, if no match is found then an incautious engine will explore it twice, once from each path from the decision point to the mutual state. In the extreme, this can lead to *super-linear behavior* (SL behavior) when the evaluation complexity, *i.e.,* the total number of explored states, is polynomial or exponential in the input length. SL behavior is also known as catastrophic backtracking.

SL behavior will only occur when a backtracking regex engine evaluates an *SL regex* on a *malign input*. A malign input has three components: a prefix string, a pump string, and a suffix string[2]. The *prefix* brings the NFA to a set of states from which redundant exploration becomes possible (*ambiguous states* [18]). Each repetition of the *pump* yields a decision point whose complete exploration contains redundant states. A final *suffix* ensures that the regex will not match the input, forcing the regex engine to explore polynomially or exponentially many redundant states during backtracking.

2.2 Example: An SL Regex in Python Core

For a detailed example of an SL regex, consider the regex represented in Figure 1a. We discovered this regex in the Python core library `difflib` (CVE-2018-1061). Malign input for this regex consists of: the empty prefix, a pump of "any whitespace", and a suffix of "any non-whitespace".

On this malign input, a backtracking regex engine will perform an $O(n^2)$ doubly-nested traversal of the input [51], as illustrated in Figure 1b. When in the first whitespace vertex WS-1, each of the N pumps is a decision point: the NFA chooses whether to remain or to advance to the second whitespace vertex WS-2 by skipping the optional '#' vertex. When the suffix causes a mismatch, the regex engine tries the other choice. As a result, the NFA will reach the WS-2 vertex N times, and each move will occur at a different NFA state (same vertex, different offset). When the NFA advances at offset $i > 1$, it will *redundantly* explore states *<WS-2, O-i>* through *<WS-2, O-N>* since these are also explored after the NFA advances at earlier offsets. Thus the engine will visit each of the *<WS-1, O-i>* states once, and each of the *<WS-2, O-j>* states j times, for a total of $(N) + (1 + 2 + ... + N) = O(N^2)$ visited states.

2.3 Using SL Behavior for ReDoS

Backtracking regex engines can be used as a denial of service vector. Regular expression denial of service (ReDoS) attacks use an SL

[2]Malign input may contain multiple pairs of prefixes and pumps.

regex query to consume the CPU of a server, reducing the number of clients it can service [23, 40]. ReDoS attacks have four requirements. **First**, the victim must use a backtracking regex engine. **Second**, the victim must use an SL regex. **Third**, the victim must evaluate (malign) user input against this regex without any resource limitations (*e.g.,* a timeout). **Fourth**, the victim must be running on the server-side. If all four conditions are met, an attacker carries out a ReDoS exploit by sending *malign input* to the victim to be evaluated against its SL regex, triggering SL behavior that wastes server resources, reducing the CPU available to other clients.

2.4 Mechanisms to Prevent SL Behavior

The practitioner community has explored three approaches to preventing SL behavior: abortive backtracking, disallowing backtracking altogether, and avoiding SL regex anti-patterns.

Abortive Backtracking. Some mainstream languages defend developers against SL regex engine behavior. The .NET framework added optional regex timeouts in 2012 [3, 53], and the PHP and Perl engines throw exceptions if they perceive too much backtracking.

Non-backtracking engines. A more radical approach is to use the linear-time regex evaluation algorithm developed by Thompson [48], though this requires deviating from the Perl-Compatible Regular expressions (PCRE) standard supported by many languages[3]. This approach was popularized by Cox in 2007 [22] and has since been adopted by Rust [14] and Go [11].

Avoiding anti-patterns. Several of the professional reference texts on regexes suggest "SL regex anti-patterns": developers should avoid nested quantifiers ("star height") [30, 32], and more generally should "watch out when...[different] parts of the [regex] can match the same text" [33]. Although this advice is vague, avoiding SL regexes is the only cross-language technique available to practitioners. Timeouts and linear-time engines are luxuries not available everywhere, so practitioners must be ready to address the worst-case behavior of the regexes in their codebases.

3 RESEARCH QUESTIONS

Our goal in this study is to understand ReDoS vulnerabilities in practice across three themes: their incidence in practice, how they can be prevented, and how they can be fixed. In particular, we focus our investigation on studying SL regexes, which can be exploited to cause ReDoS. We ask seven research questions along these themes.

First, we study the incidence of SL regexes in practice to understand to what extent ReDoS is a serious vulnerability affecting many different kinds of software projects. The answer to this investigation will help us understand the importance of ReDoS vulnerabilities.

Second, we study whether SL regex anti-patterns do in fact signal SL regexes. As we discussed in §2.4, avoiding SL regex anti-patterns is the only cross-language mechanism used in practice to prevent ReDoS, but this approach is not yet evidence-based. In this investigation we check the validity of this conventional wisdom.

Third, we study how ReDoS vulnerabilities are fixed in practice. A lack of understanding of the right strategies to fix SL regexes is a serious gap in the research literature. We bridge this gap by studying how developers are fixing ReDoS vulnerabilities in practice. This empirical understanding will allow other developers to reuse the wisdom of the experts that are already fixing ReDoS vulnerabilities.

Theme 1: Understanding the incidence of ReDoS in practice.
RQ1: How prevalent are SL regexes in practice?
RQ2: How strongly vulnerable are the SL regexes?
RQ3: Which application domains do SL regexes affect?

Theme 2: Preventing ReDoS.
RQ4: Do SL regex anti-patterns signal SL regexes?

Theme 3: Fixing ReDoS.
RQ5: How have developers fixed ReDoS vulnerabilities?
RQ6: How would developers fix ReDoS vulnerabilities if they knew all of the currently-applied approaches?
RQ7: How effective are the fixes that developers adopt?

4 THEME 1: UNDERSTANDING THE INCIDENCE OF REDOS IN PRACTICE

4.1 RQ1: How Prevalent are SL Regexes in Practice?

To date there have been a small number of reports of SL regexes leading to ReDoS in the wild, which we discuss in depth in §6. How many more remain undiscovered? In this section we present the first systematic study of the incidence of SL regexes in practice.

4.1.1 Methodology. In brief, this is how we measured the incidence of SL regexes in the wild. We used static analysis to extract all the regexes used in the Node.js and Python core libraries as well as more than half of the modules in the npm (JavaScript) and pypi (Python) registries. We applied SL regex detectors to filter for potentially-SL regexes, and concluded with a dynamic validation phase to prove that a regex was actually vulnerable.

Software. We chose *JavaScript* as our primary language of interest for two reasons. First, as others have observed [21, 25, 36], ReDoS vulnerabilities in JavaScript are particularly impactful because JavaScript frameworks use a single-threaded event-based architecture. A ReDoS attack on a Node.js server immediately reduces the throughput to zero. Second, JavaScript has a huge developer base — there are more open-source libraries for JavaScript than any other language. The registry of JavaScript modules, *npm* [12], has over 590,000 modules [26], more than double the size of the next-closest registry (Java/Maven). To gauge the generality of our results, we also studied *Python*, another popular scripting language whose *pypi* [13] registry contains 130,000 modules.

The source code in software ecosystems can be divided into the language core ("platform"), 3rd-party libraries, and applications [35]. While applications are difficult to enumerate, in modern ecosystems the language core and 3rd-party libraries are generally open-source, and 3rd-party libraries are conveniently organized in a registry that tracks metadata like where to find the module's source code. As a result, we studied the incidence of SL regexes in each language's core libraries and 3rd-party modules listed in the registries.

For each language's core, we tested each supported version. For 3rd-party libraries, we examined the master branch of every module listed in the npm and pypi registries that had a URL on which we could run `git clone`. We chose not to use the packaged version of

[3] In particular, a linear-time engine precludes general support for fundamentally super-linear features like backreferences and lookaround assertions.

modules provided by the registries because these are sometimes packed, minified, or otherwise obfuscated in ways that complicate analysis, attribution, and vulnerability reporting.

Extracting regexes. After cloning each module, we statically extracted its regexes. We cloned the latest master branch, with no history to minimize the impact on the VCS hosting service. Then we scanned it for source code based on file extensions (`.js` or `.py`). We built an abstract syntax tree (AST) from each source file, using `babylon` [6] for JavaScript files and the Python AST API for Python files. Walking the ASTs, we identified every regex declaration and extracted the pattern, skipping any uses of dynamic patterns. Excluding these dynamic patterns means our results provide lower bounds on the number of SL regexes.

Identifying SL regexes. After extracting the regexes used in each module under study, we created a mapping from unique patterns to the modules using them. We then analyzed these unique patterns.

Our SL regex identification process has a static detection phase and a dynamic validation phase. For the static *detection* phase, we queried all three of the *SL regex detectors* developed in previous work: `rxxr2` [38], `regex-static-analysis` [51], and `rexploiter` [55]. These detectors use different algorithms to report whether or not a regex may exhibit super-linear behavior, and if so will recommend malign input to trigger it. Our static phase collects each detector's opinion and produces a summary. The detectors, most frequently `regex-static-analysis`, may consume excessive time or memory in making their decision, so we limited the detectors to 5 minutes and 1GB of memory on each regex and discarded unanswered queries. These SL regex detectors are research prototypes, so they do not support all regex features nor guarantee correctness.

Our dynamic *validation* phase uses this summary to test the accuracy of each detector's prediction for the regex engine of the language of interest. The detectors follow different algorithms based on assumptions about the implementation of the regex engine, and these assumptions may or may not hold in each language of interest. To validate a detector's predicted malign input, our validator tests this malign input on the possibly-SL regex in small Node.js and Python applications we created.

This is how we identified *SL regexes*. To permit differentiating regexes by their degree of vulnerability (§4.2), we measured how long each regex took to match a sequence of malign inputs with varying numbers of pumps. We began with one pump and followed a geometric sequence with a factor of 1.1, rounding up. We tested 100 inputs, the last with 85,615 pumps, and marked the regex super-linear if the regex match took more than 10 seconds on a match, as this is far longer than a linear-time regex match would take. We stopped at 85,615 pumps for two reasons. First, this number was sufficient to cause super-linear complexity to manifest without being attributable to the overheads of enormous strings. Second, this many pumps results in malign inputs 100K-1M characters long, long enough to become potentially expensive for attackers to exploit. We distributed this analysis and ran multiple tests on each machine in parallel, dedicating one core to each test with `taskset` [1] to remove computational interference between co-located tests.

Table 1: Results of our search for SL regexes in the npm and pypi module registries. Troublingly, 1% of unique regexes were SL regexes, affecting over 10,000 modules.

Registry	Total Modules	Scanned Modules	Unique Regexes	SL Regexes	Affected Modules
npm	565,219	375,652 (66%)	349,852	3,589 (1%)	13,018 (3%)
pypi	126,304	72,750 (58%)	63,352	704 (1%)	705 (1%)

4.1.2 Results. We found that SL regexes are surprisingly common in practice. The Node.js and Python core libraries both contained SL regexes, and about 1% of all unique regexes in both npm and pypi were SL regexes. In all, 3% of npm modules and 1% of pypi modules contained at least one SL regex.

Language Core. We found one SL regex in the core libraries of Node.js (server-side JavaScript). The currently supported versions of Node.js are v4, v6, v8, and v9. We scanned the core libraries (`lib/`) of each of these versions. In v4 we identified and disclosed two SL regexes used to parse UNIX and Windows file paths. These regexes had been removed for performance reasons in v6 so the other versions of Node were not affected. This vulnerability was published as CVE-2018-7158 and fixed by the Node.js core team.

We found three SL regexes in the core libraries of Python. The currently supported versions of Python are v2 and v3. We scanned the core libraries (`Libs/`) of each of these versions. Both versions shared two SL regexes, one in `poplib` and one in `difflib`. We identified an additional vulnerability in the v2.7.14 `fpformat` library. These vulnerabilities were published as CVE-2018-1060 and CVE-2018-1061; we authored the patches.

Third-party modules. Table 1 summarizes the results of our registry analysis. We were able to clone 66% of npm (375,652 modules) and 58% of pypi (72,750 modules). In this sample of each registry we found that about 1% of the unique regexes were SL regexes (3,589 in npm, and 704 in pypi).

Figure 2 summarizes two different distributions in the npm and pypi datasets using Cumulative Distribution Functions (CDFs). The dotted lines show the distribution of the number of unique regexes in each module. We can see that more than 30% of npm and pypi modules use at least one regex, and that npm modules tend to contain more unique regexes than pypi modules do. The solid lines show the distribution of the number of modules each SL regex appears in: in the npm registry some SL regexes appear in hundreds or thousands of modules, while in the pypi registry the most ubiquitous SL regexes are only used in about 50 modules.

To give a sense of how impactful these SL regexes might be, for each module we obtained the popularity (registry downloads/month) and computed the project size based on the source files we scanned (using `cloc` [9]). Modules with SL regexes are indicated in black in Figure 3 (npm) and Figure 4 (pypi). In both registries, larger modules are more likely to contain SL regexes, and SL regexes are slightly more common in modules with lower download rates.

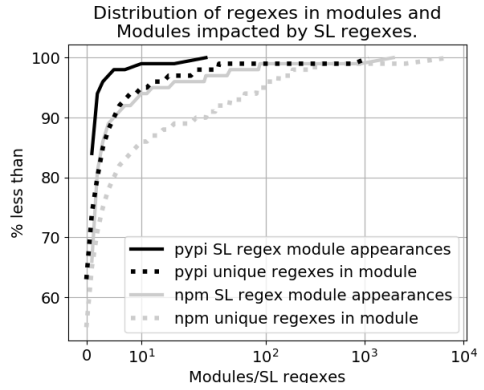


Figure 2: This figure shows two CDFs. The dotted lines indicate the distribution of the number of unique regexes used in modules, while the solid lines show the distribution of the number of modules affected by SL regexes. Note the log scale on the x-axis.

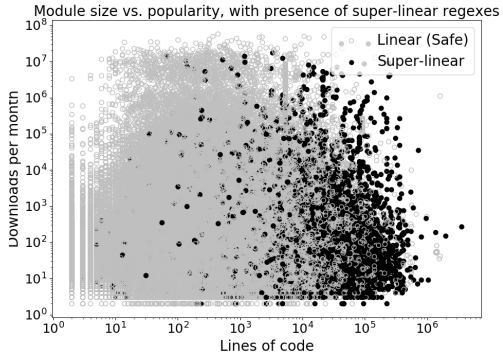


Figure 3: npm modules by size and popularity (log-log). The 13,018 modules with SL regexes are in black. Note the "trivial packages" on the left side [15].

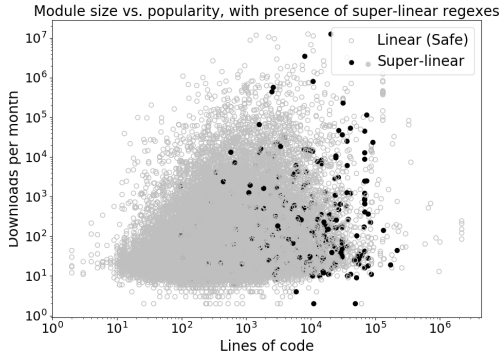


Figure 4: pypi modules by size and popularity (log-log). The 705 modules with SL regexes are in black.

4.2 RQ2: How Strongly Vulnerable are the SL Regexes?

From a developer perspective, SL regexes whose super-linear behavior manifests on shorter malign inputs are of greater concern than those only affected by longer malign inputs. Longer malign inputs could be prevented by other parts of the software stack (*e.g.*, limits on HTTP headers), while short malign inputs may only be prevented by modifications to the vulnerable software itself.

In this section we refine our definition of SL regexes, differentiating between *exponential* and *polynomial* vulnerabilities.

Table 2: This table shows the degree of vulnerabilities in the npm and pypi datasets. The polynomial vulnerabilities are further broken down by the degree of the polynomial, b, which we rounded to the nearest integer. This excluded some regexes whose polynomial degree rounded down to 1.

Degree of vulnerability	npm (3,589 vulns)	pypi (704 vulns)
Exponential $O(2^n)$	245 (7%)	41 (6%)
Polynomial $O(n^2)$	2,638 (74%)	534 (76%)
Polynomial $O(n^3)$	535 (15%)	107 (15%)
Polynomial $O(n^4)$	44 (1%)	5 (1%)
Polynomial $O(n^{b>4})$	100 (3%)	15 (2%)

4.2.1 Methodology. While the degree of vulnerability of an SL regex might be predicted by static analysis, we are not confident of the accuracy of such a prediction since it is not tied to a particular regex engine implementation. Thus, we used curve fitting to differentiate between exponential and polynomial SL regexes. As discussed in §4.1, our dynamic validation step tests the match time of the appropriate regex engine (JavaScript-V8 or Python) on a sequence of malign inputs with a geometrically increasing number of pumps. We measured the time that it took to compute each match. We then fit the time taken for different numbers of pumps against both exponential ($f(x) = ab^x$) and polynomial (power-law: $f(x) = ax^b$) curves and we chose the curve that provided the better fit by r^2 value. When the malign inputs from the different SL regex detectors resulted in different curves, we used the steepest, deadliest curve. As in §4.1.1, we distributed the work across multiple machines. As result, the multiplicative factors of the curves (a) are not comparable, but the bases or exponents (b) are.

This analysis allows us to create a hierarchy of vulnerabilities. Exponential SL regexes are more vulnerable than polynomial SL regexes, because the number of pumps (length of malign input) required to achieve noticeable delays is smaller. For the same reason, among polynomial SL regexes, those with larger b values are more vulnerable than those with smaller b values. The curve type and the b values influence the degree of vulnerability more strongly than the a values.

4.2.2 Results. A breakdown of the regexes by their degree of vulnerability is in Table 2. Exponential SL regexes were rare in both registries: only 7% of the SL regexes from npm and 6% of those from pypi were exponential. The majority of the SL regexes in both registries were polynomial, tending to $O(n^2)$ and $O(n^3)$. This finding has implications for SL regex detectors as well as for software developers.

For detecting SL regexes. The `rxxr2` REDOS [38] detector only looks for exponential SL regexes. So too do two of the SL regex antipatterns of conventional wisdom, Star Height and QAD (discussed in §5.1). These approaches to detecting SL regexes will thus miss about 90% of SL regexes.

Table 3: Proposed common semantic meanings for regexes. The examples are automatically-labeled (SL) regexes from our npm dataset. The last two columns are the number of regexes labeled with each semantic meaning in our npm and pypi datasets.

Meaning	Example	npm	pypi
Error messages	`/no such file '.+[/\\](.+)'/`	22,197	881
File names*	`/[a-zA-Z-0-9_\/-]+\.json/`	10,151	497
HTML	`/href="(.+\.css)(\?v=.+?)?"/`	8,786	2,504
URL*	`/^.+:\/\/[^\n\\]+$/`	6,986	2,048
Naming convention	`/^[$_a-z]+[$_a-z0-9-]*$/`	4,096	1,056
Source code	`/function.*?\(.*?\)\s*\{\s*/`	3,941	105
User-agent strings*	`/Chrome\/([\w\W]*?)\./`	3,135	124
Whitespace*	`/(\n\s*)+$/`	2,016	441
Number*	`/^(\d+|(\d*\.\d+))+$/`	762	238
Email*	`/^\S+@\S+\.\w+$/`	444	97
Classification rate	—	18%	13%

For working with regexes. The super-linear behavior of polynomial regexes typically manifests for malign inputs on the order of many hundreds or thousands of characters long. Such strings are often longer than any legitimate strings, as is the case for strings with many of the semantic meanings listed in Table 3 (§4.3). Thus, rejecting too-long strings before testing them against a regex would be a cheap and effective defense approach and should be considered as a best practice when writing regexes.

4.3 RQ3: Which Application Domains do SL Regexes Affect?

Regexes are used in a variety of application domains. From our own experience in writing regexes, and from a manual analysis of 400 regex uses in npm modules, we posit that developers often write regexes with one of the semantic meanings listed in Table 3. These semantic meanings may be of interest in some application domains but not others. For example, we imagine that identifying source code or naming conventions is the domain of linters and compilers, that webservers are more interested in identifying HTML and user-agent strings, and that servers or scripts may be prepared to change their behavior based on the error messages that they encounter.

4.3.1 Methodology. In this section, we describe our techniques to automatically categorize regexes into these semantic groups. We began by manually labeling the semantic meaning of 400 regex usage examples based on inspection of the regex itself as well as how it was used in the project(s) in which we found it. Although some of the regexes we encountered were obscure and their purpose could only be identified by looking for comments and other clues in the surrounding source code, it became clear to us that many regexes with the semantic meanings listed in Table 3 could be automatically classified. There were 200 unique regexes among these 400 examples, and we found that the duplicated regexes were always used with the same semantic meaning in different modules.

We developed an automatic labeling scheme that uses a combination of parsing and "meta-regexes" to label regexes based on the proposed semantic meanings. For example, here is a simplified version of our meta-regex to label regexes as describing whitespace:

```
/^\^?(\\s|\\n|\\t| |[\|\*\+\[\]\(\)]|)+\$?$/
```

This simplified regex looks for a string (regex pattern) containing only whitespace characters, as well as meta-characters that might be used to anchor the pattern ('^' and '$') or to encode varying quantities of whitespace ('+', '*', etc.).

We iteratively improved our regex labeler. In each iteration, we labeled a randomly selected subset of 10,000-30,000 regexes from our npm regex dataset. We manually examined 100 of the regexes assigned to each semantic meaning. One or more representatives of any mis-labeled regexes were added to a test suite, and the iteration was complete once the regex labeler correctly identified all the regexes in the suite.

This process resulted in a precise regex labeler for regexes that are reasonably specific. As you might expect based on how we derived it, our labeler works well for "easy to classify" regexes that restrict the input to something close to the expected language.

We refined our labeler through 17 iterations. At the conclusion of this process our test suite contained 358 regexes, and we were reasonably confident in its precision. We then applied it to our npm and pypi datasets. Irrespective of whether our list of semantic meanings for regexes is complete, it serves the goal of studying how different domains may be affected by ReDoS. We leave the search for a complete list of regex semantic meanings to future work.

4.3.2 Results. First, as summarized in Table 3, we found regexes in all of these domains in both npm and pypi. Second, some semantic meanings are more prone to being expressed with SL regexes than others. As can be seen in Figure 5, developers should be cautious when writing regexes for emails, user-agent strings, source code, and HTML.

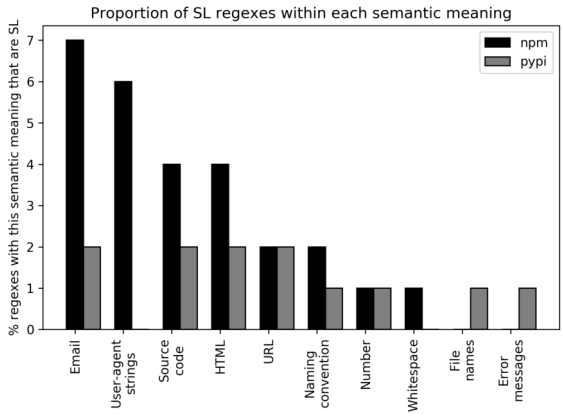


Figure 5: Percent of SL regexes from the npm and pypi datasets, within each semantic meaning.

5 THEME 2: PREVENTING REDOS

Though we used detectors to identify SL regexes in §4.1.1, these detectors are academic prototypes not ready for use in production. On the other hand, software developers already have conventional wisdom about what makes a regex super-linear. If this conventional wisdom is accurate, then publicizing it may be an effective path to preventing ReDoS by identifying and eliminating SL regexes.

5.1 RQ4: Do SL Regex Anti-Patterns Signal SL Regexes?

In §2 we mentioned several SL regex anti-patterns: avoid nested quantifiers, and avoid regexes with ambiguity. In this section we test two aspects of this conventional wisdom. First, we evaluate the extent to which these SL regex anti-patterns appear in SL regexes — are these anti-patterns a *necessary* condition for SL behavior? Second, we measure the extent to which these anti-patterns also appear in safe regexes, to determine whether these anti-patterns are a *sufficient* condition for SL behavior.

5.1.1 Methodology. **Three SL regex anti-patterns.** We know of three SL regex anti-patterns. In §2.4 we introduced the two SL regex anti-patterns discussed in reference texts on regexes. The first, star height, is discussed in many places including [30, 32, 46]. The second is rather more vague: "watch out when...[different] parts of the [regex] can match the same text" [33]. We identified two distinct ways that such ambiguity arose in our SL regex corpora, yielding three total anti-patterns.

The first anti-pattern is *star height > 1, i.e.,* nested quantifiers. This leads to SL behavior when the same string can be consumed by an inner quantifier or the outer one, as is the case for the string "a" in the regex `/(a+)+/`. In this case, the star height of two results in two choices for each pump, with worst-case exponential behavior on a mismatch. This anti-pattern is commonly used in practice through the `safe-regex` tool [46].

The second anti-pattern is a form of ambiguity that we call *Quantified Overlapping Disjunction* (QOD). An example of this anti-pattern is `/(\w|\d)+/`. Here we have a quantified disjunction (`/(...|...)+/`), whose two nodes overlap in the digits, 0-9. On a pump string of a digit there are two choices of which group to use, with worst-case exponential behavior on a mismatch.

The third anti-pattern is another form of ambiguity that we call *Quantified Overlapping Adjacency* (QOA). We gave a detailed example of this anti-pattern in §2.2: `/\s*\.#\s*$/`. The two quantified `\s*` nodes overlap, and are adjacent because one can be reached from the other by skipping the optional octothorpe. This anti-pattern has worst-case polynomial behavior on a mismatch.

Testing for the anti-patterns. We implemented tests for the presence of these anti-patterns using the `regexp-tree` regex AST generator [8][4]. To measure *star height* we traverse the AST and maintain a counter for each layer of nested quantifier: +, *, and ranges where the upper bound is at least 25[5]. To detect *QOD* we search the AST for quantified disjunctions. When we find them we enumerate the unicode ranges of each member of the disjunction and test for overlap. To detect *QOA* we search the AST for quantified nodes. From each node we walk forward looking for a reachable quantified adjacent node with an overlapping set of characters, stopping at the earliest of: a quantified overlapping node (QOA), a non-overlapping non-optional node (no QOA), or the end of the nodes (no QOA). For QOD and QOA our prototype only considers nodes containing individual quantified characters (*e.g.,* `/\d+/` or `/.*/`, not `/(ab)+/`). In keeping with the conventional wisdom, none of our tests includes a check for a mismatch-triggering suffix.

[4]The *safe-regex* tool [46] is implemented incorrectly, so we used our own implementation. We have provided a patch to the author of *safe-regex*.
[5]Groups with lower quantifications do not readily exhibit super-linear behavior.

Table 4: Utility of the SL regex anti-patterns, as measured by our tools. For each anti-pattern, we present the number of SL regexes that had this pattern in each ecosystem, and then the false positive rate. For the false positive rate we rely on the SL regex detectors (§4.1.1) as ground truth. For example, in npm 12% of the SL regexes had star height > 1, but 94% of the regexes with star height > 1 were linear-time. As some regexes have multiple anti-patterns, the final row eliminates double-counting.

Anti-pattern	Number of SL regexes		False positive rate	
	npm	pypi	npm	pypi
Star height > 1	443 (12%)	62 (2%)	94%	98%
QOD	40 (1%)	6 (1%)	97%	95%
QOA	2,548 (71%)	555 (79%)	90%	94%
Totals	2,901 (81%)	604 (86%)	91%	95%

5.1.2 Results. The results of applying our anti-pattern tests to our npm and pypi regex datasets are shown in Table 4. While we found at least one anti-pattern in most of the SL regexes (81-86%), we also found many false positives — anti-patterns in safe regexes.

Columns 2 and 3 show that the conventional wisdom of SL regex anti-patterns appears to supply the necessary conditions for SL behavior. Our tools found these anti-patterns in over 80% of the SL regexes in both ecosystems. Among the SL regexes, the (polynomial) QOA anti-pattern was more common than the (exponential) others, agreeing with our earlier finding that polynomial SL regexes are more common than exponential SL regexes (Table 2). Not all SL regexes were labeled. We manually inspected a random sample of 70 of the unlabeled npm SL regexes and confirmed that 65 of them contained one or more of these anti-patterns, using constructions too complex for our current anti-pattern test tools to detect.

However, these anti-patterns are clearly not sufficient to make an SL regex. Our tools exhibit high false positive rates; as columns 4 and 5 show, only a small fraction of the regexes with these anti-patterns were SL regexes in either ecosystem.

6 THEME 3: FIXING REDOS

6.1 RQ5: How Have Developers Fixed ReDoS Vulnerabilities?

Here we provide the first characterization of the fix approaches developers have taken when addressing ReDoS vulnerabilities. This study tells us which fix strategies developers currently use, setting the stage for a follow-up study (§6.2) of which fix strategies developers prefer. In addition, understanding the fix approaches developers generally take is a first step towards several promising research directions. For example, researchers interested in automatically repairing ReDoS vulnerabilities will benefit from knowing which types of patches developers might be willing to apply.

6.1.1 Methodology. We were interested in thorough reports describing SL regexes and how developers fixed them. We thus searched for ReDoS in security databases using the keywords "Catastrophic backtracking", "REDOS", and "Regular expression denial

Table 5: Examples of the fix strategies for an SL regex we reported in Django [2] (CVE-2018-7536). This regex detects an email, according to both the source code and our regex labeler (§4.3). The developers chose to fix this ReDoS vulnerability using the algorithm described in "Replace".

	Example
Original	`/^\S+@\S+\.\S+$/`
Trim	if 1000 < input.length: throw error else: test with existing regex
Revise	`/^[^@]+@([^\.@]+\.)+$/`
Replace*	Custom parser: (1) Exactly one @ must occur, at neither end of the string; (2) there must be a '.' to the right of the @, but not immediately so.

of service". We searched both the CVE database [10] and the Snyk.io database [43][6]. We used any reports with two properties: (1) the report used the definition of ReDoS given in §2.3; and (2) the vulnerability was fixed and the report included a link.

For each vulnerability report, we manually categorized the fix strategy the developers took. If a fix used more than one strategy (*e.g.,* both Trim and Revise), we counted it under each of the used strategies.

6.1.2 Results. **ReDoS Reports.** We identified 45 unique historic ReDoS reports (condition 1) across the CVE and Snyk.io databases. The earliest report was from 2007 and the most recent from 2018. 37 of these reports included fixes (condition 2). Three of these reports were unique to the CVE database, 27 were unique to the Snyk.io database, and 7 appeared in both databases.

Fix strategies. Three fix strategies were typical in these reports.

(1) *Trim*: Leave the regex alone, but limit the size of the input to bound the amount of backtracking.
(2) *Revise*: Change the regex.
(3) *Replace*: Replace the regex with an alternative strategy, *e.g.,* writing a custom parser or using a library.

Table 5 gives an example of each fix strategy. Only the Revise strategy was discussed in any of the reference texts on regexes we reviewed [29, 30, 32, 33].

Table 6 summarizes the results from this study, as well as the subsequent studies on new fixes (§6.2) and on fix correctness (§6.3). In the first row, we can see that developers in the historic dataset commonly Trimmed, Revised, or Replaced, each more than 20% of the time.

6.2 RQ6: How Would Developers Fix ReDoS Vulnerabilities if They Knew All of the Currently-Applied Approaches?

In §6.1 we described the three common fix strategies developers used in the historic ReDoS reports. However, we do not know whether these developers knew every strategy, and thus we cannot be sure that they preferred one strategy over another. Next, we describe the fix strategies taken by developers who were fully aware of all of the strategies.

[6]Snyk.io's database tracks vulnerabilities in popular module registries, including npm and pypi.

Table 6: Fix approaches taken to address SL regexes, in both the historic and new datasets. Examples of each approach are given in Table 5. Some of the new fixes used more than one strategy.

		Trim	Revise	Replace	Total
Historic	*Fix approach*	8	18	11	37
	Unsafe fixes	1	2	0	3
New	*Fix approach*	3	35	15	48
	Unsafe fixes	0	0	0	0

6.2.1 Methodology. To learn what fix strategies developers would take if they knew all of the options, we needed to convince a sizable group of developers to fix SL regexes. Because we felt that the maintainers of popular modules would be more likely to fix problems therein, we examined the use of SL regexes in all npm and pypi modules downloaded more than 1000 times per month (cf. Figures 3 and 4 for the effect of this filter). We filtered these modules for those whose SL regex(es) were clearly a ReDoS vector based on a manual inspection, and contacted the maintainers of those modules by email.

In our disclosures, we included a description of the vulnerability: (1) the SL regex(es) and the files in which they lay; (2) the degree of vulnerability (§4.2) for each regex; (3) each malign input, with prefix, pump, and suffix; and (4) the length of an attack string leading to a 10-second timeout on a desktop-class machine. To facilitate our experiment, we also included: (5) a description of the three fix strategies we observed in the historic data (Table 5), with links to two patches for each.

6.2.2 Results. After applying our two-stage filter, we disclosed 284 vulnerabilities across both ecosystems to the module maintainers. 48 (17%) of our disclosures have resulted in fixes so far. Prominent projects that applied fixes based on our reports include the Hapi and Django web frameworks and the MongoDB database.

The fix strategies the maintainers chose are shown in Table 6. Compared to the historic fix strategies, developers exposed to examples of all three fix strategies still preferred Revise to Trim. The use of Revise rose from 49% to 73%, while the use of Trim fell from 22% to 6%. The use of Replace remained around 30%. Clearly these developers preferred Revise when they considered all three choices.

6.3 RQ7: How Effective are the Fixes that Developers Adopt?

Any one of fix strategies in Table 5 can go awry. To **Trim**, developers must solve a Goldilocks problem: trim too short and valid input will be rejected, trim too long and the vulnerability will remain. To **Revise**, developers must craft a linear-time regex that matches a language close enough to the original that their APIs continue to work. Lastly, to **Replace**, developers must write a parser for the input that matches an equivalent or related language.

In this study we examine the correctness of developers' fixes.

6.3.1 Methodology. Here is our fix safety classification scheme. We called a **Trim** fix unsafe if the maximum allowed input length can still trigger a noticeable slowdown. We compared the input limit to the lengths of malign inputs derived using the SL regex identification procedure from §4.1. We called a **Revise** fix unsafe if

it was labeled vulnerable by our SL regex identification procedure. We called a **Replace** fix unsafe if the replacement logic was super-linear in complexity based on manual inspection.

6.3.2 Results. Our findings for the effectiveness of the historic and new fixes are summarized in Table 6. Several of the historic fixes were incorrect. The new fixes were uniformly correct (nearly all developers asked us to review their fixes before publishing their changes).

Trim. 1 of the 8 historic Trim fixes was unsafe. The initial choice of length limit was too generous and the regex remained vulnerable for two years before this was discovered and the length limit lowered.

Revise. 2 of the 18 historic fixes resulted in a revised, but still SL, regex. One of these was replaced before our study. As a testament to the effectiveness of our approach, we discovered the other in §4.1 and disclosed it in §6.2 before performing this portion of our study.

Replace. We manually inspected the fixes that used the Replace strategy to gauge their complexity. All appeared sound, relying on one or more linear scans of the input.

Testing their fixes. Regardless of the fix strategy, developers did not usually include test cases for their changes. In the historic dataset, 8 of the 37 fixes included tests. In the new dataset, 18 of the 48 fixes included tests.

7 DISCUSSION AND RECOMMENDATIONS

We believe it is clear from our findings that ReDoS is not a niche concern but a potentially common security vulnerability. Given the number, variety, and ubiquity of SL regexes that we found, we believe that developers should not be left to their own devices.

Make regex engines less prone to SL behavior. We suggest that language developers work with regex engine developers to provide application developers with reasonable guarantees about the worst-case performance of their regexes. This guarantee might be on the computational complexity of the operation, as Rust and Go offer, though doing so restricts the range of regex features to which developers have access[7]. The guarantee might instead be about the total amount of backtracking experienced (as in Perl and PHP), or about the total amount of time that might be spent in a regex query (as the .NET framework optionally supports).

Degrees of vulnerability. Differentiating between exponential and polynomial SL regexes gives developers insight into valid fix approaches. Trimming is a possible fix strategy for polynomial SL regexes, but not for exponential ones. But make no mistake, polynomial vulnerabilities can be just as disastrous. There is little difference in the cost for attackers to send malign inputs of 100 characters or 10,000, so long as they accomplish their aim of denying service to legitimate users.

Experiences fixing SL regexes. In addition to the 48 fixes from module maintainers, we submitted 9 fixes when maintainers asked us for help. Our own experiences may illuminate some of the factors that developers will consider when selecting a fix strategy, though we believe this too is a promising direction for future research.

[7]In our npm and pypi datasets, however, these features are rarely used. 338,065 of the unique npm regexes (97%) and 58,800 of the unique pypi regexes (93%) use only linear-time features.

The fix strategy we selected (1 Trim, 9 Revise, 2 Replace, with 3 overlaps) depended on both whether the SL regex was exponential or polynomial, and how identifiable the language of the regex was. When a regex was exponential or was polynomial with a large degree, the vulnerability would manifest on short malign input. We fixed these by Revising, aided by visualizations from the `regexper` tool [4] to understand the original language and study the source of the SL behavior. When the SL behavior was less severe (*e.g.*, quadratic), we considered both Revise and Trim. When we could discern the language described by the regex, we favored Revise, but when the regex's language was unclear or many regexes were applied to the same input (*e.g.*, parsing a user agent string), Trim was an attractive alternative. We felt an aversion to Replace because it felt overly verbose.

Libraries. We were surprised by the variety of regexes we found with the same semantic meaning (§4.3). Surely we do not need 6,986 different regexes to parse URLs, nor 444 different regexes to parse emails, especially not when hundreds of these variations exhibit SL behavior. We therefore recommend developers make greater use of libraries for parsing common constructs like those with the semantic meanings indicated with a '*' in Table 3. Along these lines, it would also be helpful if RFCs included linear-time regexes to parse key fields and protocols.

Conventional Wisdom. Our findings in §5.1 (Table 4) give nuance to the conventional wisdom on SL regex anti-patterns. Though Star Height, QOD, and QOA were present in nearly all SL regexes in our datasets, they were also present in ten times as many linear-time regexes as measured by our tools. This finding speaks to the value of using SL regex detectors. However, the anti-patterns do give insight into the root cause of an SL regex, and can be used with the Revise fix strategy.

Why might SL regexes be more pollutive in npm than in pypi? We thought the difference between the npm and pypi "SL regex module appearances" curves (solid lines in Figure 2) was striking. Why might the most ubiquitous SL regexes pollute only 50 modules in pypi but hundreds in npm? We think the multi-module appearances of SL regexes in npm can be attributed to three causes. (1) *Copy/pasting* useful regexes from places like StackOverflow. We found several examples of SL email regexes originating on StackOverflow. See for example the exponential SL regex of [7], which has 1900 upvotes. A study of the regexes on StackOverflow and their intersection with our ecosystem-scale datasets would be interesting follow-up work. (2) *Software bundling*, because of disincentives in the JavaScript community to having many explicit dependencies. In one case, we identified 43 modules whose npm artifacts contained the source of another module with an SL regex. (3) Many JavaScript libraries wish to be *context-agnostic*, and excerpt core Node.js libraries to ensure that they are always available. For example, one of the SL path-parsing regexes from Node v4 appeared in over 2,000 npm modules.

8 THREATS TO VALIDITY

Construct Validity. One threat is that we used automated SL regex detectors to identify SL regexes. Our study may thus be affected by

incorrectly-labeled regexes. We address false positives by dynamically confirming the report from the SL regex detectors (§4.1.1). False negatives are also possible: we report only the SL regexes that can be detected by existing techniques (*e.g.,* none of them considers the use of inherently super-linear features like backreferences). This means that the SL regexes we identified represent a *lower bound* on the number of such regexes in practice.

Another threat is that we do not identify all the possible application domains in which regexes could be applied. Our goal in studying RQ3 was to understand whether SL regexes appear across application domains, and whether different application domains are affected differently by them. Our precise but otherwise potentially incomplete set of application domains still allowed us to answer these questions in the affirmative.

Finally, an SL regex is only one of the criteria for a ReDoS attack (§2.3). We focused on identifying ReDoS vulnerabilities, and did not confirm that they are exploitable; we did not perform taint analysis to confirm that malign input could reach these regexes, nor did we attempt to differentiate between modules intended for the server side vs. the client side.

Internal Validity. We developed several novel analyses to answer our research questions. Incorrect implementations of the regex semantic meaning labeler (§4.3) and the anti-pattern tests (§5.1) would skew our findings. To address the threat to our regex labeler, we validated its precision over 17 iterations. Our anti-pattern tests identified the use of anti-patterns in over 80% of the SL regexes, indicating agreement with the SL regex detectors in our ensemble, and agreeing with the intuition that anti-patterns are a necessary condition for super-linear behavior.

External Validity. A threat to external validity concerns whether our findings will hold for other ecosystems and scenarios. We addressed this threat by studying two popular programming languages with large ecosystems. As the general theme of our findings was consistent across these ecosystems, we expect our results to generalize to other ecosystems as well.

9 RELATED WORK

Here is a brief history of ReDoS. Crosby first suggested that regexes with large complexity could be a denial of service vector [23], as a precursor to his influential work with Wallach on algorithmic complexity attacks [24]. The first CVE report of ReDoS appeared in 2007 (CVE-2007-2026), and the notion was popularized by OWASP and Checkmarx in 2009 [40]. Two primitive SL regex detectors were released in the early 2010s: Microsoft's `SDL Regex Fuzzer` tool used input fuzzing to try to trigger super-linear behavior, while substack's `safe-regex` used the star height anti-pattern [46]. These detectors were followed by a succession of more rigorous academic works on SL regex detection: Kirrage, Rathnayake, and Thielecke [34] and Rathnayake and Thielecke [38] developed `rxxr2` in 2013-2014, Weideman et al. released `regex-static-analysis` in 2016 [51, 52] and Wustholz et al. published `rexploiter` in 2017 [55][8]. Our work takes the logical next step: we measured the extent to which SL regexes occur in real-world software and examined the adoption and effectiveness of repairs.

[8]Concurrent work from Shen et al. demonstrates an SL regex detector based on genetic search algorithms [41].

In terms of ecosystem-scale ReDoS analyses, the closest work to ours is industrial, not academic. In 2014, Liftsecurity.io performed an ecosystem-scale study of SL regexes in npm [5]. They relied on `safe-regex` to scan 100,000 modules and only identified 56 SL regexes affecting 120 modules. Their much smaller incidence rate is not surprising — `safe-regex`'s Star Height heuristic will not capture thousands of polynomial SL regexes (Table 4).

Staicu and Pradel recently demonstrated mappings from SL regexes in npm modules to ReDoS vulnerabilities in hundreds of popular websites [45], suggesting that the myriad SL regexes we found indicate many other ReDoS vulnerabilities.

An interesting line of work from van der Merwe, Weideman, and Berglund proposes automatic regex revising techniques to replace SL regexes with equivalent safe ones [49]. This work is not yet fully developed but promises to provide developers with a useful tool to address SL regexes. We note that these authors restricted themselves to revisions that would match the exact same language, while developers rarely did so in the fixes we studied. We suggest combining this automatic revision approach with an understanding of semantic meanings (§4.3) and anti-patterns (§5.1) as a promising direction for future research.

More generally, the study of regexes from a software engineering perspective was pioneered by Chapman and Stolee. They studied the use of regexes in a small sample of Python applications [19], and our study of modules and core libraries complements their work. With Wang, they have also explored possible factors affecting regex comprehension [20]. Lastly, Wang and Stolee recently found that regexes exhibit poor test coverage [50], which may contribute to the incidence of SL regexes we report.

10 REPRODUCIBILITY

An artifact containing our regex datasets and our analysis code is available at https://doi.org/10.5281/zenodo.1294300.

11 CONCLUSION

We believe nearly every practicing software developer has used regular expressions. As it turns out, many developers have also written super-linear regexes and introduced performance or security concerns in doing so. We found thousands of super-linear regexes in our analyses of the Node.js (JavaScript) and Python ecosystems, affecting over 10,000 modules as well as the core libraries of Node.js and Python.

We have found that ReDoS is not a niche concern, but rather a common security vulnerability. As such, it merits significant additional investment from researchers and practitioners. Much work remains: in the short term, to gauge developer awareness and improve educational resources, and in the long term to implement and evaluate effective prevention and resolution mechanisms.

ACKNOWLEDGMENTS

We thank the ESEC/FSE reviewers for their feedback. K. Yavine and D. Grander of Snyk.io did yeoman's work assisting us in disclosing ReDoS vulnerabilities. E.R. Williamson, A. Kazerouni, J. Phillips, and the VT Systems Reading Group gave helpful feedback on ideas and versions of this paper. This work was supported in part by the National Science Foundation, under grant CNS-1814430, and a Google Faculty Research Award.

REFERENCES

[1] 2004. taskset – set or retrieve a process's CPU affinity. https://web.archive.org/web/20180801003855/https://linux.die.net/man/1/taskset.

[2] 2005. Django: The web framework for perfectionists with deadlines. https://web.archive.org/web/20180801003925/https://www.djangoproject.com.

[3] 2012. What's new in the .NET Framework 4.5. https://web.archive.org/web/20180801003332/https://docs.microsoft.com/en-us/dotnet/framework/whats-new/index.

[4] 2014. Regexper. https://web.archive.org/web/20180801004409/https://regexper.com/.

[5] 2014. Regular Expression DoS and Node.js. https://web.archive.org/web/20170131192028/https://blog.liftsecurity.io/2014/11/03/regular-expression-dos-and-node.js.

[6] 2017. Babylon: Babylon is a JavaScript parser used in Babel. http://web.archive.org/web/20171231170138/https://github.com/babel/babel/tree/master/packages/babylon.

[7] 2017. How to validate an email address using a regular expression? https://web.archive.org/web/20180801004019/https://stackoverflow.com/questions/201323/how-to-validate-an-email-address-using-a-regular-expression.

[8] 2017. regexp-tree: Regular expressions processor in JavaScript. https://web.archive.org/web/20180801004201/https://github.com/DmitrySoshnikov/regexp-tree.

[9] 2018. cloc: Count Lines of Code. https://web.archive.org/web/20180801003246/https://github.com/AlDanial/cloc.

[10] 2018. Common Vulnerabilities and Exposures. https://cve.mitre.org.

[11] 2018. Go documentation: regexp. https://web.archive.org/web/20180801003600/https://golang.org/pkg/regexp/.

[12] 2018. npm. https://web.archive.org/web/20180801003712/https://www.npmjs.com.

[13] 2018. PyPI – the Python Package Index. https://web.archive.org/web/20180801003833/https://pypi.org/.

[14] 2018. Rust documentation: regex. https://web.archive.org/web/20180801003203/https://docs.rs/regex/1.0.2/regex/.

[15] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In *Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/3106237.3106267

[16] Alasdair Allan. 2012. *Learning iOS Programming: From Xcode to App Store*. O'Reilly Media.

[17] Adam Baldwin. 2016. Regular Expression Denial of Service affecting Express.js. http://web.archive.org/web/20170116160113/https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43.

[18] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. *EPTCS: Automata and Formal Languages 2014* 151 (2014), 109–123. https://doi.org/10.4204/EPTCS.151.7

[19] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2931037.2931073

[20] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring Regular Expression Comprehension. In *Automated Software Engineering (ASE)*.

[21] Checkmarx. 2016. The Node.js Highway - Attacks are at Full Throttle. In *BlackHat USA*. https://www.youtube.com/watch?v=-HzCUZDLXTc

[22] Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). https://swtch.com/~rsc/regexp/regexp1.html

[23] Scott Crosby. 2003. Denial of service through regular expressions. *USENIX Security work in progress report* (2003).

[24] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security*.

[25] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *USENIX Security Symposium (USENIX Security)*.

[26] Erik DeBill. 2010. Module Counts. http://web.archive.org/web/20180114183225/http://www.modulecounts.com/.

[27] Stack Exchange. 2016. Outage Postmortem. http://web.archive.org/web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[28] Maximiliano Firtman. 2013. *Programming the Mobile Web: Reaching Users on iPhone, Android, BlackBerry, Windows Phone, and more*. O'Reilly Media.

[29] Michael Fitzgerald. 2012. *Introducing regular expressions*. O'Reilly Media, Inc.

[30] Jeffrey EF Friedl. 2002. *Mastering regular expressions*. O'Reilly Media, Inc.

[31] Patrice Godefroid. 1995. *Partial-Order Methods for the Verification of Concurrent Systems*. Ph.D. Dissertation. University of Liege. https://doi.org/10.1007/3-540-60761-7

[32] Jan Goyvaerts. 2006. *Regular Expressions: The Complete Tutorial*. Lulu Press.

[33] Jan Goyvaerts and Steven Levithan. 2012. *Regular expressions cookbook*. O'Reilly Media, Inc.

[34] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *International Conference on Network and System Security (NSS)*. 35–148.

[35] Konstantinos Manikas and Klaus Marius Hansen. 2013. Software ecosystems-A systematic literature review. *Journal of Systems and Software* 86, 5 (2013), 1294–1306. https://doi.org/10.1016/j.jss.2012.12.026

[36] A Ojamaa and K Duuna. 2012. Assessing the security of Node.js platform. In *7th International Conference for Internet Technology and Secured Transactions (ICITST)*.

[37] Theoolos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Computer and Communications Security (CCS)*. https://doi.org/10.1145/3133956.3134073

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

[39] Randy J. Ray and Pavel Kulchenko. 2002. *Programming Web Services with Perl*. O'Reilly Media.

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

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

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

[43] Snyk.io. 2018. Vulnerability DB. http://web.archive.org/web/20180801010155/https://snyk.io/vuln.

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

[45] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium (USENIX Security)*. https://www.npmjs.com/package/safe-regexhttp://mp.binaervarianz.de/ReDoS_TR_Dec2017.pdf

[46] substack. 2013. safe-regex. https://web.archive.org/web/20180801003748/https://github.com/substack/safe-regex.

[47] Bryan Sullivan. 2010. *Security Briefs - Regular Expression Denial of Service Attacks and Defenses*. Technical Report. https://msdn.microsoft.com/en-us/magazine/ff646973.aspx

[48] Ken Thompson. 1968. Regular Expression Search Algorithm. *Communications of the ACM (CACM)* (1968). https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

[49] Brink Van Der Merwe, Nicolaas Weideman, and Martin Berglund. 2017. Turning Evil Regexes Harmless. In *SAICSIT*. https://doi.org/10.1145/3129416.3129440

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

[51] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. 2016. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 9705. 322–334.

[52] Nicolaas Hendrik Weideman. 2017. *Static Analysis of Regular Expressions*. MS. Stellenbosch University.

[53] Wikipedia contributors. 2018. .NET Framework version history — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=.NET_Framework_version_history&oldid=851937435. [Online; accessed 1-August-2018].

[54] Wikipedia contributors. 2018. Regular expression — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=852858998. [Online; accessed 1-August-2018].

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