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

James C. Davis  
Virginia Tech  
Francisco Servant  
Virginia Tech

Christy A. Coghlan  
Virginia Tech  
Dongyoon Lee  
Virginia Tech

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, or syntax highlighting [16], and are a standard component of programming textbooks [18, 27, 36].

However, in recent years, it has been reported that some regexes can lead to a security risk known as regular expression denial of service (ReDoS) [24]. ReDoS causes server-side software to overload the point of the server denying service to clients (DoS) as a result of an attacker-controlled input being evaluated against a vulnerable regex that it almost matches. The almost-match causes a backtracking-based regex engine to try many ways of traversing the string — polynomially or even exponentially many ways. Due to the computational complexity of this attempt, the server overloads and cannot service clients. A high degree of backtracking is also known as catastrophic backtracking.

Regexp engines are susceptible to ReDoS when their implementation is backtracking-based. Unfortunately, most modern, popular programming languages use such an engine, e.g., JavaScript-V8 (Node.js), Python, Java, C++, C#, Mono, PHP, Perl, and Ruby.

To the best of our knowledge, the link between catastrophic backtracking and the risk of Regular Expression Denial of Service (ReDoS) was first proposed by Crosby in 2003 [24]. Since then, some researchers have proposed various techniques for detecting vulnerable regular expressions [31, 34, 35, 41, 44, 46]. Additionally, some practitioners have introduced language-level mechanisms to limit the impact of catastrophic backtracking in some programming languages, e.g., by timing out long backtracking (.NET [17]) or by using an alternative matching algorithm (Rust [14], Go [7]). Practitioners also proposed ReDoS anti-patterns as ways to prevent ReDoS.

However, the existing research still does not provide a good understanding of how serious the ReDoS problem is in practice. We do not know how prevalent vulnerable regexes are in software, nor how strongly they can slow software down. Furthermore, we do not have a clear understanding of the ways in which vulnerable regexes can be prevented or fixed. To the extent of our knowledge, ReDoS anti-patterns have not yet been empirically evaluated. Also, language-level solutions to limit catastrophic backtracking are only useful if one happens to use the languages that support them. Thus,
developers still have no cross-language accepted mechanisms to address catastrophic backtracking.

In this paper, we perform the first empirical study to understand the impact of ReDoS in practice as well as the mechanisms that could be used to prevent and fix it. In our study, we analyze the ecosystems of two of the most popular programming languages to understand the incidence of vulnerable regexes. Our study covers the Node.js (JavaScript) and Python core libraries, as well as more than 50% of the modules in their package registries — npm [8] and pypi [9]. We also use this data to empirically evaluate ReDoS anti-patterns as a way to prevent ReDoS. Additionally, we study these repositories to understand the fixes that developers provide for vulnerable regexes.

In the results of our studies, we found that vulnerable 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 module registries, including popular modules with millions of downloads per month. We found over 4,000 unique vulnerable 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 vulnerable regexes in 284 modules, of which 48 have been fixed so far.

We also found that the conventional wisdom of ReDoS anti-patterns does describe a necessary condition for vulnerable regexes, but not a sufficient one — the vulnerable regexes contain anti-patterns, but few of the anti-pattern appearances lead to vulnerabilities. In terms of the fixes that developers provide for vulnerable regular expressions, we found that developers fix them in three ways: trimming the input, revising the regex, or replacing it with alternative logic. Among these options, revising the regex was the most popular, regardless of whether developers were previously aware of all these options. Fixing vulnerable regexes is relatively difficult, since not all the applied fixes fully resolved the ReDoS vulnerability as the developers intended.

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 vulnerable regular expressions in two prominent software ecosystems — Node.js and Python.
- We provide an empirical understanding of the relationship between ReDoS anti-patterns that conventional wisdom recommends avoiding and the incidence of vulnerable regexes (§5.1).
- We provide an empirical understanding of the strategies that developers use to fix vulnerable regexes (§6).

2 BACKGROUND

In this section, we describe how regular expression denial of service (ReDoS) attacks work, as well as the existing mechanisms so far to prevent them.

2.1 How ReDoS Attacks Work

At a high level, ReDoS attacks use an expensive regex query to overload the CPU of a server-side process, reducing the number of connections it can service. ReDoS attacks have two requirements. First, the regex engine used by the victim must use backtracking to evaluate inputs against regexes. Unfortunately, this requirement is met by most modern and popular programming languages — e.g., JavaScript-V8 (Node.js), Python, Java, C++-11, C#-Mono, PHP, Perl, and Ruby. Second, the victim must evaluate attacker input against a vulnerable regex [24, 37].

If both conditions are met, the attacker carries out a ReDoS exploit by sending malignant input to the victim. A malignant input forces the regex engine to explore a vast search space to check whether the input matches the regex — which correspondingly causes the CPU overload. Because each step in the search requires backtracking, this phenomenon is known as catastrophic backtracking.

2.1.1 Catastrophic Backtracking. At the core of most regex engines is a backtracking-based search algorithm. Regex engines accept a regex describing a language, and an input to be tested for membership in this language. A backtracking-based regex engine constructs a non-deterministic finite automaton (NFA) from the regex and then simulates the NFA on the input [38]. To simulate non-determinism, whenever the engine makes a choice it pushes the current NFA state onto a stack of backtracking points. If a mismatch occurs, the engine backtracks recursively, trying alternative decisions until it either finds a match or exhausts the set of backtracking points.

Catastrophic backtracking is the term for the result when a regex engine has to explore an enormous number of states before it can declare a match or mismatch for an input. Typically the number of explored states is polynomial (commonly $O(n^2)$ or $O(n^3)$) or exponential in the input length.

2.1.2 Vulnerable Regexes and Malign Inputs. A regex is vulnerable to catastrophic backtracking when its evaluation may require the regex engine simulating the NFA to make a large amount of non-deterministic choices on malignant input.

A malignant input has three components: a prefix string, a pump string, and a suffix string. The prefix brings the NFA to a set of ambiguous states [19]. These states are called ambiguous because on a special subsequent input (the pump) the NFA can move from one of these states to another by more than one path. If it sees such an input, a backtracking regex engine will try one path and save backtracking points for the others. By repeating the pump, a malignant input expands the search space by forcing repeated choices, building up the stack of backtracking states. A final suffix causes a mismatch, triggering polynomial or exponential backtracking through the stack of saved states.

2.2 A real-world example: ReDoS in Python

We illustrate ReDoS by describing a vulnerable regex that we found in the Python core library difflib. This regex is represented in Figure 1. Its ambiguous states are the two “whitespace” nodes. The components of the malignant input are: an empty prefix, a pump of “any whitespace character”, and a suffix of “any non-whitespace character”.

When it encounters each pump, the regex engine makes a choice: to stay in the first whitespace node or to advance to the second

---

1 One constructs malign input by concatenating the components as follows: $MI = \text{prefix} \cdot \text{pump} \cdot \text{suffix}$. The more times a malign input repeats the pump string, the more potent it is.
by skipping the optional '#’ node. Suppose the regex engine’s implementation causes it to choose first to advance when possible. When the suffix causes a mismatch, the regex engine revisits all of its choices and tries the other one, which was to stay in the first node. Because this happens for every whitespace character in the string, the search space is equivalent to an $O(n^2)$ doubly-nested traversal of the string.

Because this regex could be used by a Python-based server to process user input, it represents a ReDoS vulnerability. Vulnerable regexes with $O(n^2)$ complexity are generally fast on short malign inputs. The complexity of this specific regex manifests on malign inputs of around 1,500 characters.

2.3 Mechanisms to Prevent Catastrophic Backtracking

The practitioner community has explored three approaches to preventing catastrophic backtracking: abortive backtracking, disallowing backtracking altogether, and avoiding ReDoS anti-patterns. The only proposed cross-language approach to prevent catastrophic backtracking is to avoid anti-patterns that are expected to be particularly risky. However, to the extent of our knowledge, this recommendation has not yet been empirically evaluated.

Abortive Backtracking. Some mainstream languages defend developers against catastrophic backtracking. The .NET framework introduced optional regex timeouts in 2012 [7, 17], while the engines in PHP and Perl will throw exceptions if they perceive too much backtracking.

Non-backtracking engines. A more radical approach is to restrict the supported regex features to those that can be implemented with a linear-time scan of the input string, using algorithms developed by Thompson [42]. This approach was popularized by Cox in the mid 2000’s [23] and has since been adopted by Rust [14] and Go [?].

Avoiding anti-patterns. Several of the professional reference texts on regexes suggest “ReDoS anti-patterns”: developers should avoid nested quantifiers (“star height”) [28, 29], and more generally should “watch out when...[different] parts of the [regex] can match the same text” [30].

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. Particularly, we focus our investigation on studying vulnerable regexes that could be exploited to cause ReDoS. For that goal, we ask seven research questions along these three themes.

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

Second, we study whether ReDoS anti-patterns do in fact signal vulnerable regexes. As we discussed in §2.3, avoiding ReDoS anti-patterns is the only cross-language mechanism to try to prevent ReDoS, but these patterns are not yet backed on evidence. In this part of our study we check the validity of this conventional wisdom.

Third, we study how ReDoS vulnerabilities are fixed in practice. Another serious gap in the research literature is an understanding of the right strategies to fix vulnerable regexes. 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.

We pose seven research questions along these three themes:

4 THEME 1: UNDERSTANDING THE INCIDENCE OF REDOS IN PRACTICE

4.1 RQ1: How prevalent are vulnerable regexes in practice?

To date there have been a handful of anecdotal reports of vulnerable regexes leading to ReDoS in the wild, which we will discuss in depth in §6. In this section we present the first systematic study of the incidence of vulnerable regexes in practice.

4.1.1 Methodology. In brief, this is how we measured the incidence of vulnerable 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 ReDoS detectors to filter for potentially-vulnerable regexes, and concluded with a dynamic validation phase to prove that a regex was actually vulnerable.

Which software? We chose JavaScript as our primary language of interest for two reasons. First, as others have observed [22, 26, 33], 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 libraries, npm [8], has over 590,000 modules [6], 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 [9] registry has 130,000 modules.

The source code in software ecosystems can be divided into the language core ("platform"), 3rd-party libraries, and applications [32]. 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 vulnerable 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 each module listed in the npm and pypi registries that listed URLs on which we could run git clone. We chose not to use the packaged version of modules provided by the registries because these are sometimes packed, minified, or otherwise obfuscated in ways that complicate analysis 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). For each source file, we built an abstract syntax tree (AST). For npm we used babylon [13], while for pypi we used the Python AST API. Walking the ASTs, we identified every regex declaration and extracted the pattern, skipping any uses of dynamic patterns. Excluding these dynamic patterns mean our results provide lower bounds on the number of vulnerable regexes.

**Identifying vulnerable regexes.** After extracting the regexes used in each module under study, we reduced this list to a unique set of patterns mapped to the modules using them. We then analyzed these unique patterns.

Our vulnerable regex identification process has a static detection phase and a dynamic validation phase. For the static detection phase, we queried all three of the catastrophic backtracking detectors ("ReDoS detectors") developed in previous work: rxxr2 [35], regex-static-analysis [44], and rexploiter [46]. Each of these detectors uses a different algorithm to report whether or not a regex is vulnerable to catastrophic backtracking, and if vulnerable will recommend malign input (prefix, pump, and suffix) that will trigger catastrophic backtracking. 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.

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-vulnerable regex in small Node.js and Python applications we created for this purpose.

This is how we identified vulnerable 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 vulnerable if the regex match took more than 10 seconds on a match. 10 seconds is an eternity for a busy server. 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 ran this analysis in parallel, so we dedicated one core to each test with taskset [15] to remove interference between co-located tests.

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

**Language Core.** In JavaScript, our interest in ReDoS led us to test the core libraries for Node.js (server-side JavaScript), located in Node.js’s lib/ directory. The currently supported versions of Node.js are v4, v6, v8, and v9. In v4 we identified and disclosed two vulnerable regexes used to parse UNIX and Windows file paths 3. These regexes had been removed for performance reasons in v6 so the other versions of Node were not affected.

The currently supported versions of Python are v2 and v3. We scanned the core libraries (lib/ and stdlib/) of Python v2.7.14 and Python v3.6.4. Both versions shared two vulnerable regexes, one in poplib and one in difflib. We identified an additional vulnerability in the v2.7.14 fpformat library. Our patches for these three regexes were merged.

**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 about 1% of the unique regexes to be vulnerable (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 generally have far more regexes than pypi modules do. The solid lines show the distribution of the number of modules each vulnerable regex appears in. They tell us that in the npm registry some vulnerable

<table>
<thead>
<tr>
<th>Registry</th>
<th>Total Modules</th>
<th>Scanned Modules</th>
<th>Unique Regexes</th>
<th>Vuln. Regexes</th>
<th>Affected Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>npm</td>
<td>355,219</td>
<td>375,622</td>
<td>349,852</td>
<td>5,389</td>
<td>13,018</td>
</tr>
<tr>
<td>pypi</td>
<td>126,304</td>
<td>72,750</td>
<td>63,352</td>
<td>704</td>
<td>705</td>
</tr>
</tbody>
</table>

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

3These and several other vulnerabilities were assigned CVE numbers, which we omit for blinding purposes.
regexes appear in hundreds or thousands of modules, while in the pypi registry the most ubiquitous vulnerable regexes are only used in about 50 modules.

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 vulnerable regexes. Note the log scale on the x-axis.

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

Figure 3: npm modules by size and popularity (log-log). The 13,018 modules with vulnerable regexes are in black.

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

### Table 2: Degree of Vulnerability

<table>
<thead>
<tr>
<th>Degree of Vulnerability</th>
<th>npm (3,589 vulns)</th>
<th>pypi (704 vulns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>245 (7%)</td>
<td>41 (6%)</td>
</tr>
<tr>
<td>Power</td>
<td>3,344 (93%)</td>
<td>663 (94%)</td>
</tr>
<tr>
<td>$n^2$</td>
<td>2,638 (74%)</td>
<td>534 (76%)</td>
</tr>
<tr>
<td>$n^3$</td>
<td>535 (15%)</td>
<td>107 (15%)</td>
</tr>
<tr>
<td>$n^4$</td>
<td>44 (1%)</td>
<td>5 (1%)</td>
</tr>
<tr>
<td>$b &gt; 4$</td>
<td>100 (3%)</td>
<td>15 (2%)</td>
</tr>
</tbody>
</table>

#### 4.2 RQ2: How strongly vulnerable are the vulnerable regexes?

From a developer perspective, vulnerable regexes whose super-linear behavior manifests on shorter malign inputs are of greater concern than those that time out on 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 vulnerable regexes, differentiating between exponential and polynomial vulnerabilities.

#### 4.2.1 Methodology

We used curve fitting to differentiate between exponential and polynomial vulnerabilities. As discussed in §4.1, our dynamic validation step tests the match time of the regex engine 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 power ($f(x) = ax^b$) curves and we chose the curve that provided the best fit based on the $r^2$ values.

This analysis allows us to create a hierarchy of vulnerabilities. Regexes with exponential curves are more vulnerable than those with power curves, because the number of pumps (length of malign input) required to achieve noticeable delays is much smaller. For the same reason, regexes with power curves and 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. Sometimes two detectors found malign inputs on which super-linear behavior manifested at different rates. In such cases we used the deadlier curve, i.e., the steeper curve.

#### 4.2.2 Results

A breakdown of the regexes by their degree of vulnerability is in Table 2. Exponential vulnerabilities were relatively rare in both registries: only 7% of the vulnerable npm regexes and 6% of the vulnerable pypi regexes were exponential. The majority of the vulnerabilities in both registries were polynomial, tending to $O(n^2)$ and $O(n^3)$.
Table 3: Proposed common semantic meanings for regexes. The examples are automatically-labeled vulnerable 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.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Example</th>
<th>npm</th>
<th>pypi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error messages</td>
<td>/no such file .+(\{(.+)})/</td>
<td>22,197</td>
<td>881</td>
</tr>
<tr>
<td>File names*</td>
<td>/[a-zA-Z-0-9_]+.[a-zA-Z0-9-]*$/</td>
<td>10,151</td>
<td>497</td>
</tr>
<tr>
<td>HTML</td>
<td>/\bhref=(:.+)\b$/</td>
<td>8,786</td>
<td>2,504</td>
</tr>
<tr>
<td>URL*</td>
<td>/\b+:%/\b(&quot;\b[\b]$$/</td>
<td>6,986</td>
<td>2,048</td>
</tr>
<tr>
<td>Naming convention</td>
<td>/[^$%[]()+-.]+[a-zA-Z-0-9_]+.[a-zA-Z-0-9_]+.$/</td>
<td>4,096</td>
<td>1,056</td>
</tr>
<tr>
<td>Source code</td>
<td>/\bfunction.<em>,?\b(\b.</em>,?\b)$$/</td>
<td>3,941</td>
<td>105</td>
</tr>
<tr>
<td>User-agent strings*</td>
<td>/\bChrome(\b[[]\b[]]\b)$$/</td>
<td>3,135</td>
<td>124</td>
</tr>
<tr>
<td>Whitespace*</td>
<td>/\b%\b(\b/\b\b%\b)$$/</td>
<td>2,016</td>
<td>441</td>
</tr>
<tr>
<td>Number*</td>
<td>/\b[\d]+\b/</td>
<td>762</td>
<td>238</td>
</tr>
<tr>
<td>Email*</td>
<td>/\b&lt;\b[\d]+\b\b/</td>
<td>444</td>
<td>97</td>
</tr>
</tbody>
</table>

Table 2 tells us that the vast majority of vulnerable regexes are polynomially so, not exponentially so. This has implications for detecting vulnerable regexes as well as for working with regexes. For detecting vulnerable regexes, the regexr REDoS detector only looks for exponential vulnerabilities. So too do two of the ReDoS anti-patterns of conventional wisdom, Star Height and QAD (discussed in §5.1). These approaches to detecting vulnerable regexes will thus miss about 90% of vulnerable regexes. For working with regexes, the super-linear behavior of polynomial regexes typically manifests for Postfix union of strings 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. 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: Do vulnerable regular expressions affect different application domains?

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 may 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 uses 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, 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 regexes and more complex parsing to label regexes based on the proposed semantic meanings. For example, here is a simplified version of the “meta-regex” we used to label regexes as describing whitespace:

```
/\%\b\(\b/\b\%\b\)\$$/
```

This simplified regex looks for a string 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 labeling 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 labeling works well for “easy to classify” regexes that restrict the input to something close to the expected language. We refined our labeling 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 are affected by ReDoS. We leave the search for a complete list of regex semantic meanings to future work.

4.3.2 Results. We found two things: regexes in all of these domains were vulnerable in one or both of npm and pypi, and semantic meanings are more prone to vulnerabilities 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. We provide more insight into the root causes of these vulnerabilities in the next section.

5 THEME 2: PREVENTING REDOS

5.1 RQ4: Do conventional-wisdom ReDoS anti-patterns signal vulnerable regexes?

In §2 we mentioned several “ReDoS 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 ReDoS anti-patterns appear in vulnerable regexes — are these anti-patterns a necessary condition for ReDoS? Second, we measure the extent to which these anti-patterns also appear in safe regexes, to determine whether the use of these anti-patterns is a sufficient condition for ReDoS.

5.1.1 Methodology.

Three ReDoS anti-patterns. We know of three ReDoS anti-patterns.

In §3 we introduced the two ReDoS anti-patterns discussed in
reference texts on regexes. The first, star height, is discussed in many places including [28, 29, 40]. The second is rather more vague “watch out when...[different] parts of the [regex] can match the same text” [30]. We identified two distinct ways that such ambiguity arose in our vulnerable regex corpora, yielding three total anti-patterns.

The first anti-pattern is star height > 1 (nested quantifiers). Star height is an anti-pattern when the same (pump) string can be consumed in the inner quantification or the outer one. /\(a\)+$/ is an example of such a regex. In this case, the star height of two results in two choices for each pump and yields an exponential degree of backtracking.

The second anti-pattern is one form of ambiguity, Quantified Overlapping Disjunction (QOD). An example of this anti-pattern is /\(\w|\d\)+$/ . The two groups in the disjunction overlap in the numbers 0 through 9, so on a pump string of a number there are two choices of which group to use, again resulting in an exponential degree of backtracking.

The third anti-pattern is another form of ambiguity, Quantified Overlapping Adjacency (QOA). An example of this anti-pattern is /\(d\)+\d\+/ . The two quantified \d+ nodes overlap on any number, and they are adjacent because you can reach one from the other by skipping the optional period (decimal point). On a pump string of a number this anti-pattern results in a polynomial amount of backtracking, as described in §2.

Testing for the anti-patterns. We implemented tests for the presence of each of these anti-patterns using the regexp-tree regex AST generator [12]. To detect star height we traversed the AST and maintained a counter for each layer of nested quantifier. For quantifiers we used +, *, and ranges where the upper bound is at least 25. To detect QOD we traversed the AST looking for quantified disjunctions. When we found them we enumerated the unicode ranges of each member of the disjunction and searched for overlap. To detect QOA we traversed the AST examining “runs” of adjacent nodes. We began one run at each quantification and searched for a quantified adjacent node with an overlapping set of characters. So, from each quantification we iterated over subsequent nodes, stopping at the earliest of the end of the nodes (no QOA), a non-overlapping non-optional node (no QOA), or a quantified overlapping node (QOA).

We then applied these tests to our npm and pypi regex datasets.

5.1.2 Results. The results of applying our anti-pattern tests to our npm and pypi regex datasets are shown in Table 4.

First, columns 2 and 4 show that the conventional wisdom of ReDoS anti-patterns appears to explain the necessary conditions for ReDoS vulnerabilities. At least 80% of the vulnerable regexes in our npm dataset (85% for pypi) had one or more of these anti-patterns, and from manual inspection we believe most of the remainder also use these anti-patterns. Among the vulnerable regexes, the QOA anti-pattern was much more common than the Star Height or QOD anti-patterns, agreeing with our finding that polynomial vulnerabilities were much more common than exponential vulnerabilities.

Our anti-pattern tests cannot yet label particularly complex regexes, so 20% of the npm vulnerable regexes and 14% of the pypi vulnerable regexes were unlabeled. We manually inspected a random sample of 70 (10%) of the unlabeled npm regexes, and confirmed that at least 65 of them contained one or more of these anti-patterns.

Second, however, these anti-patterns are clearly not sufficient to make a regex vulnerable. As columns 3 and 5 show, only a small fraction of the regexes with these anti-patterns were vulnerable in either ecosystem.

6 THEME 3: FIXING REDOS

6.1 RQ5: How have developers fixed ReDoS vulnerabilities?

Table 4: Percent of vulnerable regexes containing each anti-pattern, out of (a) all vulnerable regexes, and (b) all regexes containing the anti-pattern. Some regexes contain more than one anti-pattern.

<table>
<thead>
<tr>
<th>ReDoS anti-pattern</th>
<th>npm vulnerable regexes</th>
<th>Among regexes with pattern</th>
<th>pypi vulnerable regexes</th>
<th>Among regexes with pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star height &gt; 1</td>
<td>12%</td>
<td>6%</td>
<td>9%</td>
<td>2%</td>
</tr>
<tr>
<td>QOD</td>
<td>1%</td>
<td>3%</td>
<td>1%</td>
<td>5%</td>
</tr>
<tr>
<td>QOA</td>
<td>71%</td>
<td>10%</td>
<td>79%</td>
<td>6%</td>
</tr>
</tbody>
</table>

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) on 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 vulnerable regexes and how developers fixed them. We thus searched for ReDoS in security databases using the keywords “Catastrophic backtracking”, “REDOS”, and “Regular expression
Table 5: Examples of the fix strategies for a vulnerability we reported in Django [3]. Django’s regex was detecting an email, according to both our regex labeler (§4.3) and the surrounding source code. The developers chose to fix this vulnerability using the algorithm described in “Replace”.

<table>
<thead>
<tr>
<th>Example</th>
<th>Origina</th>
<th>Trim</th>
<th>Revise</th>
<th>Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>/*\S+@\S+.\S+$/</td>
<td>if 1000 &lt; input.length: throw error else: test with existing regex</td>
<td>/*[^@]+@([^@]+.)+$/</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

Table 6: Fix approaches taken to address vulnerable 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.

<table>
<thead>
<tr>
<th>Fix approaches</th>
<th>Trim</th>
<th>Revise</th>
<th>Replace</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Historic</td>
<td>8</td>
<td>18</td>
<td>11</td>
<td>37</td>
</tr>
<tr>
<td>Unsafe fixes</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>New</td>
<td>3</td>
<td>35</td>
<td>15</td>
<td>48</td>
</tr>
<tr>
<td>Unsafe fixes</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 vulnerable regexes. Because we felt that the maintainers of popular modules would be more likely to fix problems therein, we examined the use of vulnerable 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 use of vulnerable regexes was clearly a possible ReDoS vector based on a cursory manual inspection, and contacted the maintainers of these modules by email.

This is the information we provided in our disclosures². First, we included a description of the vulnerability: (1) the vulnerable 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. We also offered to validate any fixes the developers proposed, a service of which almost all availed themselves.

6.2.2 Results. After applying our two-stage filter, we disclosed 284 vulnerabilities from these two 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 web framework, the MongoDB database, and the Django web framework.

The fix strategies they 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 developers preferred Revise when they knew all three choices. Rather than speculate on why developers made the choices they did, in §7 we share our personal experiences repairing 9 ReDoS vulnerabilities, which may shed some light on the matter.

6.3 RQ7: How effective are the fixes that developers adopt?

Any one of fix strategies in Table 5 can go awry. If they Trim, developers must solve a Goldilocks problem: too short and valid input will be rejected, too long and the vulnerability will remain. If they Revise, developers must craft a safe regex that matches a
language close enough to the original that their APIs continue to work. Lastly, if they 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 how we labeled each fix as safe or unsafe. 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 vulnerable regex identification procedure from §4.1. We called a Revise fix unsafe if it was labeled vulnerable by our vulnerable 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 because nearly all developers asked us to review their fixes before publishing their code on GitHub. We cannot disclose the details because we did not obtain their consent.

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 vulnerable, 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

Make regex engines less prone to catastrophic backtracking. We believe it is clear from our findings that ReDoS is not a niche concern but a common security vulnerability. Left to their own devices, developers will continue to write vulnerable regexes.

Given the number, variety, and ubiquity of vulnerable regexes, we believe that developers should not be left on their own. 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. 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).

6 Degrees of vulnerability. Differentiating between exponential and polynomial vulnerabilities gives developers insight into valid fix approaches. Trimming is a possible fix strategy for polynomial vulnerabilities, but not for exponential ones. But make no mistake, polynomial vulnerabilities can be just as disastrous as exponential ones. 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 vulnerable patterns. In addition to the 48 fixes from module maintainers, we submitted 9 fixes when developers 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.

The fix strategy we selected (1 Trim, 9 Revise, 2 Replace, with 3 overlaps) depended on both whether the vulnerability was exponential or polynomial, and how identifiable the language of the regex was. When vulnerabilities were exponential or were 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 to understand the original language and study the source of the catastrophic backtracking. When the vulnerability was less severe (e.g., quadratic), we considered both Revise and Trim. When we could discern the regex’s language, 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 inelegant.

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 are vulnerable to catastrophic backtracking. We therefore recommend developers make greater use of libraries of regexes, especially for the semantic meanings indicated with a * in Table 3. Such libraries would also simplify the process of checking regexes for catastrophic backtracking. Along these lines, it would also be helpful if RFCs included safe regexes to parse key fields and protocols.

Conventional Wisdom. Our findings in §5.1 (Table 4) give nuance to the conventional wisdom on ReDoS anti-patterns. Though Star Height, QOD, and QOA were present in nearly all vulnerable regexes in our datasets, they were also present in ten times as many safe regexes. The ReDoS anti-patterns are simply ineffective ways to identify vulnerable regexes. This result speaks to the value of using ReDoS detectors that analyze the NFA rather than just the pattern’s AST. However, the anti-patterns do give insight into the root causes of the vulnerable regexes, and can be used with the Revise fix strategy.

Why might vulnerable regexes be more pollutive in npm than in pypi? We thought the difference between the npm and pypi “vulnerable regex module appearances” curves (solid lines in Figure 2) was striking. Why might the most ubiquitous vulnerable regexes pollute only 50 modules in pypi but hundreds in npm? We think the multi-module appearances of vulnerable regexes in npm can be attributed to three causes. (1) Copy/pasting useful
regexes from places like StackOverflow. We found several examples of vulnerable email regexes originating on StackOverflow. See for example [5], with 1900 upvotes and a 300-character blow-up, and [4]. A study of the regexes of 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 a vulnerable 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 vulnerable path-parsing regexes from Node v4 appeared in over 2,000 npm modules.

8 THREATS TO VALIDITY

Construct Validity. A threat to construct validity is the fact that we used automated ReDoS detectors to identify instances of ReDoS in practice. As a result of this decision, our study may be affected by regex instances incorrectly being labeled as vulnerable. This decision may also limit our study to finding only the ReDoS instances that can be detected by such techniques (e.g., none of them considers the use of inherently non-linear-time features like back-references). We address the first threat by double-checking whether each identified regex is actually vulnerable by running the attack scenario in a controlled environment (see Section 4.1.1). Also, the second threat only means that our identified vulnerable regexes represent a lower bound of the incidence of ReDoS in practices — they may be even more prevalent than we found.

Another threat to construct validity is the fact 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 vulnerable 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 find the answer to both questions: Yes.

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. We believe that for our anti-pattern tests the proof is in the pudding — our tests identified the use of anti-patterns in 80% of the vulnerable regexes, which agrees with the intuition that anti-patterns are a necessary condition for ReDoS vulnerabilities.

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 of the most popular programming languages with the largest ecosystems. For that reason, we expect the findings in our study to be resilient in other ecosystems, and that they will benefit large developer communities.

9 RELATED WORK

Here is a brief history of ReDoS. Crosby first suggested that regexes with large complexity could be a DoS vector, as a precursor to his influential work on algorithmic complexity attacks [24, 25]. The first CVE report of ReDoS appeared in 2007 [2], and the notion was popularized by OWASP and Checkmarx in 2009 [37]. Two primitive ReDoS detectors were released in the early 2010s: Microsoft’s SDL Regex Fuzzer tool used input fuzzing to try to trigger catastrophic backtracking, and substack’s star height-based safe-regex was released in 2013 [40]. These detectors were followed by a succession of more rigorous academic works on ReDoS detection: Kirrage, Rathnayake, and Thielecke [31] and Rathnayake and Thielecke [35] developed rxxr2 in 2013-2014. Weideman et al. released regex-static-analysis in 2016 [44, 45] and Wustholz et al. published exploiter in 2017 [46]. Our work takes the logical next step: we measured the extent to which ReDoS occurs in real-world software and examined the effectiveness and adoption of defenses.

In terms of ecosystem-scale ReDoS analyses, the closest work to ours is industrial, not academic. Liftsecurity.io has a brief blog post from 2014 describing their ecosystem-scale study of vulnerable regexes in npm [11]. They relied on safe-regex to scan 100,000 modules and only identified 56 vulnerable regexes affecting 120 modules. Their much smaller incidence rate is not surprising — we demonstrated that safe-regex’s Star Height heuristic will not capture thousands of vulnerable poly-time regexes (Table 4).

An interesting line of work from van der Merwe, Weideman, and Berglund proposes automatic regex revising techniques to replace vulnerable regexes with equivalent safe ones [43]. This work is not yet fully developed but promises to provide developers with a useful tool to address vulnerable 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 [20], and our study of modules and core libraries complements this work. With Wang, they also explored possible factors affecting regex comprehension [21].

10 CONCLUSION

We believe nearly every practicing software developer has used regexes. As it turns out, many developers have also written vulnerable regexes and introduced performance or security concerns in doing so. We found thousands of vulnerable regexes in ecosystem-scale analyses of npm (JavaScript) and pypi (Python), including vulnerabilities in 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 the need for education, and in the long term to implement and evaluate effective prevention and resolution mechanisms.

We included a reproducibility package with this paper submission. Because of the security-sensitive nature of this work, we are releasing our source code (about 10KLoC) but not our datasets.
REFERENCES

[15] [n. d.]. taskset - set or retrieve a process's CPU affinity. https://linux.die.net/man/1/taskset. ([n. d.]).