2023

Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies

Sk Adnan Hassan
Virginia Tech, skadnan@vt.edu

Zainab Aamir
Stony Brook University, zaamir@cs.stonybrook.edu

Dongyoon Lee
Stony Brook University, dongyoon@cs.stonybrook.edu

James C. Davis
Purdue University, davisjam@purdue.edu

Francisco Servant
University of Malaga, fservant@uma.es

Hassan, Sk Adnan; Aamir, Zainab; Lee, Dongyoon; Davis, James C.; and Servant, Francisco, "Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies" (2023). *Department of Electrical and Computer Engineering Faculty Publications.* Paper 161.
https://docs.lib.purdue.edu/ecepubs/161

Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies

Sk Adnan Hassan*
Virginia Tech
Blacksburg, VA, USA
skadnan@vt.edu

Zainab Aamir, Dongyoon Lee
Stony Brook University
Stony Brook, NY, USA
{*zaamir, dongyoon*}*@cs.stonybrook.edu*

James C. Davis
Purdue University
West Lafayette, IN, USA
davisjam@purdue.edu

Francisco Servant†
University of Málaga
Málaga, Spain
fservant@uma.es

Abstract—**Regular expressions are used for diverse purposes, including input validation and firewalls. Unfortunately, they can also lead to a security vulnerability called ReDoS (Regular Expression Denial of Service), caused by a super-linear worst-case execution time during regex matching. Due to the severity and prevalence of ReDoS, past work proposed automatic tools to detect and fix regexes. Although these tools were evaluated in automatic experiments, their usability has not yet been studied; usability has not been a focus of prior work. Our insight is that the usability of existing tools to detect and fix regexes will improve if we complement them with anti-patterns and fix strategies of vulnerable regexes.**

We developed novel anti-patterns for vulnerable regexes, and a collection of fix strategies to fix them. We derived our anti-patterns and fix strategies from a novel theory of regex infinite ambiguity — a necessary condition for regexes vulnerable to ReDoS. We proved the soundness and completeness of our theory. We evaluated the effectiveness of our anti-patterns, both in an automatic experiment and when applied manually. Then, we evaluated how much our anti-patterns and fix strategies improve developers' understanding of the outcome of detection and fixing tools. Our evaluation found that our anti-patterns were effective over a large dataset of regexes (N=209,188): 100% precision and 99% recall, improving the state of the art 50% precision and 87% recall. Our anti-patterns were also more effective than the state of the art when applied manually (N=20): 100% developers applied them effectively vs. 50% for the state of the art. Finally, our anti-patterns and fix strategies increased developers' understanding using automatic tools (N=9): from median *"Very weakly"* to median *"Strongly"* when detecting vulnerabilities, and from median *"Very weakly"* to median *"Very strongly"* when fixing them.

Index Terms—**Regular expression denial of service, Usability**

1. Introduction

Regular expressions (*regexes*) are a tool for text processing [1], [2]. Regexes are used across the system stack [3]–[6], including in security tasks such as input validation [7], [8] and web application firewalls [9], [10]. Unfortunately, regexes can themselves cause a security vulnerability because of the high worst-case time complexity of backtracking-based regex engine implementations. This algorithmic complexity vulnerability is known as *Regular Expression Denial of Service (ReDoS)* [11], [12]. For example, ReDoS caused service outages at Stack Overflow in 2016 [13] and at Cloudflare in 2019 [14]. Researchers report hundreds of vulnerable regexes in the software supply chain [2] and in live web services [15], [16].

Many approaches have been proposed to address the ReDoS problem. Our work builds on those that try to detect and fix regexes. In this vein, some researchers characterized vulnerable regexes into anti-patterns for manual use by developers [2]. Others proposed tools to automatically detect [17]–[26] or fix [27]–[30] vulnerable regexes. All of these approaches have been evaluated solely via automatic experiments. Their *usability* has not been studied, jeopardizing their impact in practice [31] — 95% of developers reject tools when they cannot understand the results [31].

The goal of this paper is to improve the usability of existing ReDoS defenses. Our insight is that the usability of existing tools to detect and fix regexes will improve if we complement them with *anti-patterns* and *fix strategies* of vulnerable regexes. We specifically aim to improve developer understanding of the outcome of the tools.

For this goal, we developed novel anti-patterns for vulnerable regexes, and a collection of fix strategies to fix them. We derive our anti-patterns and fix strategies from our novel theory of regex infinite ambiguity (IA). Our theory characterizes a fundamental component of vulnerable regexes: their infinitely ambiguous (IA) region. The IA region is what the state of the art anti-patterns characterize [2], what many detection tools detect, *e.g.,* [18], [20], and what developers often fix in vulnerable regexes [2]. We refer to regexes with an IA region as IA regexes. Our anti-patterns and fix strategies complement existing detection and fixing tools, (1) by helping developers better understand the IA region of the vulnerable regex detected by the tool; and (2) by providing understandable fix strategies in addition to the ones proposed by fixing tools.

Our evaluation proceeded in four phases: proving our

[*] Sk Adnan Hassan is currently employed at Walmart Inc.

[†] *Some work performed while at Virginia Tech, U.S.A., and Universidad Rey Juan Carlos, Madrid, Spain.*

theory, and then running three experiments. *First*, we formally proved the IA theory on which our anti-patterns are based (§4 and §B). *Second*, since we deliberately introduced inaccuracy in our anti-patterns in favor of simplicity, we evaluated their effectiveness in an automatic experiment. We compared our anti-patterns to the state-of-the-art ones over a large dataset of regexes (§7). *Third*, since low usability may lower effectiveness in manual use [31], we also evaluated the effectiveness of our anti-patterns when applied manually. We compared our anti-patterns to the state-of-the-art ones in a human-subjects experiment, simulating a context in which developers often prefer manual techniques [31], *e.g.,* when tools disrupt developer workflow, such as in regex composition or when working with simple regexes (§8). *Fourth*, for more complex tasks, developers may prefer to use automatic tools. So, we also evaluated how our anti-patterns and fix strategies complement the usage of existing automatic tools by improving their usability. In a second human-subjects experiment (§9), we measured if our anti-patterns improve the understanding of the outcome of tools to (a) detect and (b) fix vulnerable regexes (§9). To the best of our knowledge, this is the first study of the usability of anti-patterns or tools to detect or fix vulnerable regexes when applied by humans.

Our evaluation provided multiple findings. *First*, our underlying theory of regex infinite ambiguity was sound and complete. *Second*, our anti-patterns provided higher effectiveness (100% precision, 99% recall) than the state of the art anti-patterns [2] (50% precision, 87% recall) over a dataset of 209,188 real-world regexes [32]. *Third*, novice and intermediate developers (100% of 20 studied) increased their effectiveness at identifying IA in regexes over 5 different regex tasks, improving from a success rate of 50% to a rate of 100%. *Fourth*, the 9 expert developers who used our anti-patterns to complement detection tools increased their understanding of what makes a detected regex vulnerable: from median *"Very weakly"* to median *"Strongly"*. Similarly, when using our fix strategies to complement fixing tools, they increased their understanding of what makes the resulting fixed regex not vulnerable: from median *"Very weakly"* to median *"Very strongly"*.

This paper provides the following contributions:

1) A sound and complete theory of regex infinite ambiguity (§4).
2) Derived from this theory, IA anti-patterns (§5) and IA fix strategies (§6).
3) A quantitative evaluation of the comprehensiveness of our IA anti-patterns over the largest dataset of real-world regexes, showing that they capture IA effectively in a wide proportion of them (§7).
4) The first usability evaluation of characterizations of vulnerable regexes, showing that our IA anti-patterns were usable enough for novice developers to apply effectively in the absence of tools (§8).
5) The first usability evaluation of tools to detect and fix vulnerable regexes, showing that our IA anti-patterns and IA fix strategies improved their usability by improving their understanding (§9).

Our paper provides a replication package [33] (see §A).

2. Background

Regular Expressions (Regexes) and Ambiguity:

Regexes. Kleene proposed regular expressions as a notation to specify a language, *i.e.,* a set of strings [34]. With a finite alphabet of terminal symbols, Σ, and metacharacters, '$|$', '$\cdot$', and '$*$', the regular expression syntax is [35]:

$$R \rightarrow \phi \;\Big|\; \epsilon \;\Big|\; \sigma \;\Big|\; R_1|R_2 \;\Big|\; R_1{\cdot}R_2 \;\Big|\; R_1*$$

where ϕ denotes the empty language; ϵ is the empty string; the characters $\sigma \in \Sigma$ are terminal symbols; $R_1|R_2$ alternates; $R_1{\cdot}R_2$ concatenates; and $R*$ repeats. The language function $L : R \rightarrow 2^{\Sigma^*}$ gives semantics:

$$\begin{array}{ll} L(\phi) = \phi & L(R_1|R_2) = L(R_1) \cup L(R_2) \\ L(\epsilon) = \{\epsilon\} & L(R_1{\cdot}R_2) = L(R_1){\cdot}L(R_2) \\ L(\sigma) = \{\sigma\} & L(R*) = L(R)* \end{array}$$

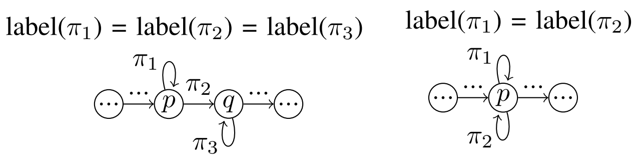
These semantics apply to Kleene's regexes, and extend to "syntax sugar" notations such as character ranges `[a-c]`. In practice, regexes may include non-regular features such as lookaround assertions, backreferences, and possessive quantifiers [36]. These features are used in less than 10% of real-world regexes [1], [32], [37]. We therefore focus on the common case of Kleene-regular regexes, denoted *K-regexes*.

Regex Ambiguity. The regex language semantics allow membership to be checked with a parser. A regex is *ambiguous* if there is a string in its language that can be matched by more than one parse tree [25], [38]. For example, the regex `a|a` can parse the input "*a*" in two ways, *i.e.,* yielding two parse trees, one using the left `a` and one the right.

For K-regexes, a regex match is equivalent to simulating an input on a corresponding non-deterministic finite automaton (NFA) [39]. To simplify discussion, we will reason about regex ambiguity over an equivalent, ambiguity-preserving, ϵ-free NFA [19], [40]. From the NFA perspective, a regex is ambiguous if there is a string that can be accepted along multiple paths of this NFA.

Infinitely Ambiguous (IA) Regex. Regexes have various degrees of ambiguity [41], [42]: no ambiguity; finite (bounded regardless of input length); or infinite in input length. Infinite ambiguity (IA) leads to super-linear time complexity in some parsing algorithms (*e.g.,* backtracking) [18], [20]. A regex is infinitely ambiguous if it has an infinitely ambiguous (IA) region (equivalently, an NFA section), *i.e.,* a region with the *infinite-degree-of-ambiguity (IDA)* property [40]. Given an ϵ-free finite automaton A, necessary and sufficient conditions for A to be infinitely ambiguous are given by Weber & Seidl [40].

IDA can be of two types: (1) polynomially IDA (PDA), and (2) exponentially IDA (EDA). Figure 1(a) illustrates a *polynomially IDA (PDA)* section in a regex's NFA. A substring $label(\pi_i)$ can be matched in the loop π_1 at node p, the path π_2 from p to q, or in the loop π_3 at q. For example, consider the regex `a*a*` for an input "$aa...a$" of length N. As any two partitions of the input can be matched with the first `a*` and second `a*`, there are N matching paths.


label(π_1) = label(π_2) = label(π_3)
π_1
π_2
π_3
p
q
label(π_1) = label(π_2)
π_1
π_2
p

(a) PDA (or IDA_d)

(b) EDA

Figure 1: Illustration of Polynomial and Exponential Degree of Ambiguity (PDA, EDA) in the NFA [40]. We say that (p, π_2, q) is a transition from state p to state q via label(π_2) [20].

Figure 1(b) illustrates an *exponentially IDA (EDA)* section in a regex's NFA. A substring $label(\pi_i)$ can be matched in either of two loops π_1 or π_2 at node p. Consider the example regex `(a|a)*`. Each 'a' of the input "$a...a$" can be matched by either the upper or lower loop, and thus the total number of matching paths becomes 2^N.

Regex-Based Denial of Service (ReDoS): Regex-based Denial of Service (ReDoS) [11] is a security vulnerability — an algorithmic complexity attack [11] by which a web service's computational resources are diverted from legitimate client interactions into an expensive regex match, degrading its quality of service. Following Davis *et al.* [43], ReDoS involves three Conditions:

(C1) a *backtracking regex engine* used in evaluation, and
(C2) a *vulnerable regex*, applied to evaluate
(C3) a *malign input*.

C1-Backtracking Regex Engine. Many regex engines (*e.g.,* versions of PHP, Perl, JavaScript, Java, Python, Ruby, and C#) use a backtracking search algorithm, *e.g.,* Spencer's [44], to answer regex queries [32], [45].

C2-Vulnerable Regex. A vulnerable regex is an IA regex whose NFA has a *prefix* region, followed by an IA region (either PDA or EDA), followed by a *suffix* region [20]. The IA region is a necessary component and the root cause of the regex's vulnerability. The prefix must be considered to reach this IA region, and the suffix must typically lead to a mismatch in order to trigger backtracking.

C3-Malign Input. An attacker-controlled malign input triggers the super-linear behavior of a vulnerable regex by driving the backtracking engine into evaluating a polynomially or exponentially large number of possible NFA paths. The exploration exhausts computational resources [17].

Threat model. We suppose the following threat model for ReDoS, aligned with the common use of regexes for input sanitization in web software [1], [20], [46]. The victim's regex engine uses a backtracking regex engine (ReDoS Condition 1), which is common for many server-side programming languages. The victim uses a regex (C2) to sanitize attacker-controlled input (C3).

ReDoS in practice. Davis *et al.* reported two high-profile examples of ReDoS affecting millions of users [43], [47]. In §C we note growing ReDoS CVEs from 2010 to present.

3. Related Work

Empirical measurements of ReDoS in practice: Although the ReDoS attack was proposed twenty years ago by Crosby and Wallach [11], [48], researchers have only recently attempted to estimate its impact. In 2018, Davis *et al.* reported that vulnerable regexes were present in many popular open-source software modules, and that engineers struggled to fix them [2]; in 2019, they observed that these regexes displayed super-linear behavior in the built-in regex engines used in most mainstream programming languages [32], [37]. Concurrently, Staicu & Pradel showed that 10% of Node.js-based web services were vulnerable to ReDoS due to their use of vulnerable npm modules [15]. In 2022, Barlas *et al.* studied the impact of ReDoS in live web services [16]. Even in non-backtracking engines, Turoňová *et al.* observed the impact of ReDoS [49]. These findings motivated further research into the ReDoS problem.

Characterizations to Manually Detect Vulnerable Regexes: In past work, Davis *et al.* characterized the IA region of vulnerable regexes with anti-patterns, although with a high false positive rate [2]. Brabrand & Thomsen's theories precisely identify *un*ambiguous regexes, but treat all others as suspect, including both IA regexes and merely finitely ambiguous (*i.e.,* non-vulnerable) regexes [38]. [1]

In contrast with Davis *et al.*'s anti-patterns, we provide a theoretical grounding to formally capture their limitations (§4) and thus provide higher precision and recall (§7). We also evaluate their usability when applied manually by humans (§8). In contrast with Brabrand & Thomsen's, our theory distinguishes between finite and infinite ambiguity, enabling developers to distinguish between likely-unproblematic (non-IA) and problematic (IA) regexes.

Finally, other characterizations of vulnerable regexes exist, but they were not proposed to be applied manually by humans. Instead, they follow the models used by automatic detection tools, *e.g.,* expressed as finite automata [18], [20] (see Figure 1). Contrasting with these other characterizations, we designed ours to be consumed by humans. Our approach uses the modality of the regex language—the representation that developers understand best [50], [51].

Tools to Automatically Detect Vulnerable Regexes: Berglund *et al.* defined a prioritized type of NFA to simulate a backtracking engine in Java and decide if a regex could show super-linear behavior [17]. Weideman *et al.* also use a prioritized NFA to find IDA in it [18], [19]. Wustholz *et al.* also looks for the IDA pattern in the NFA and computes an attack automaton that produces attack input strings [20] . Liu *et al.* adds support for modeling and analyzing less common regex features, *e.g.,* set operations [21]. Li *et al.* prescribed five vulnerability patterns, although without theoretical validation [52].

Others statically analyze different representations of the regex for vulnerability. Kirrage *et al.* analyze an abstract

[1] Finite ambiguity could cause ReDoS for complex regexes [49] or when resources are limited (*e.g.,* a low-power device like a Raspberry Pi).

evaluation tree of the regex [22]. Rathnayake *et al.* look for exponential branching in the regex evaluation tree [23]. Sugiyama *et al.* analyzes the size of a tree transducer for the regex [24]. Finally, Sulzmann *et al.* use Brzozowski derivatives to create a finite state transducer to generate parse trees and minimal counter-examples [25].

Still other approaches detect vulnerable regexes using dynamic analysis. Shen *et al.* and McLaughlin *et al.* proposed search algorithms to find inputs with super-linear matching time [26] [53]. More general algorithmic complexity detectors, *e.g.,* [54]–[58], can also be extended to detect ReDoS.

Vulnerable regex detection tools have been evaluated for effectiveness, but not for usability. Our anti-patterns complement these tools by improving developer understanding of the outcome of their detection (§9).

Tools to Automatically Fix Vulnerable Regexes: These approaches offer trade-offs for the fixed regex, in: semantic similarity, (perceived) readability, and support for uncommon features. Van der Merwe *et al.* presented a modified flow algorithm to convert an ambiguous K-regex into an equivalent unambiguous one [27], with perfect semantic equivalence, but lower readability. More recently, Li *et al.* proposed an approach to fix vulnerable regexes with deterministic regex constraints to avoid regex ambiguity [29]. Chida & Terauchi proposed a "Programming By Example" approach that supports K-regexes, lookarounds, capture groups, and backreferences [59]. Both approaches use a human in the loop to provide good examples [29], [59]. Finally, Claver *et al.* [30] proposed a synthesis-based approach that they evaluate with synthetic regexes.

These tools have been evaluated for effectiveness, but not for usability. Our fix strategies complement these tools by improving developer' understanding of the fix (§9).

Non-regex-based Workarounds:

Recovering From ReDoS. After a system containing a ReDoS vulnerability is deployed, it is possible to detect and mitigate ReDoS attacks. Bai *et al.* proposed a ReDoS-specific approach, applying deep learning to detect and sandbox attack strings [60]. Atre *et al.* proposed using adversarial scheduling to mitigate adversarial complexity attacks [61]. Approaches that detect anomalous resource utilization, *e.g.,* time [62], CPU [57], or application-level concepts [63], can also mitigate ReDoS. These approaches reduce the impact of ReDoS, but do not remove the root cause.

Changing the regex engine. There are both classic and more recent alternatives to the exponential-time backtracking regex algorithm. The earliest published regex matching algorithms, by Brzozowski in 1964 [64] and Thompson in 1968 [65], offer linear-time guarantees. There are production-grade implementations of Thompson's approach, notably RE2 [66] and the engines in Rust [67] and Go [68]. Microsoft has considered Brzozowski's approach for .NET [69], as well as deterministic [70], [71] or hybrid [72] matching strategies. However, programming language maintainers have been slow to adopt these algorithms because of the risk of regression and the limited support for non-regular regex features [43].

4. Theory of Regex Infinite Ambiguity

Here we introduce an existing theory of regex ambiguity [38], discuss its limitations, and present our novel theorems.

Recalling §2, a regex with an infinite degree of ambiguity (IA) [40] has the necessary condition for super-linear regex behavior [2], [11], [48]. Though the NFA-level conditions for IA regexes (namely PDA and EDA regions) are well known [40], we lack characterizations in terms of regex syntax and semantics. We provide such a description to support developers assessing or composing regexes.

Preliminaries: Brabrand & Thomsen [38] developed the state of the art description of regex-level ambiguity. They introduced an overlap operator, $\bowtie$, between two languages $L(R_1)$ and $L(R_2)$. The set $L(R_1) \bowtie L(R_2)$ contains the ambiguity-inducing strings that can be parsed in multiple ways across $L(R_1)$ and $L(R_2)$. More formally, with $X = L(R_1)$ and $Y = L(R_2)$,

$$X \bowtie Y = \{xay \mid x, y \in \Sigma^* \wedge a \in \Sigma^+ s.t.\ x, xa \in X \wedge ay, y \in Y\}$$

Using this operator, Theorem 0 summarizes their findings.

Theorem 0 (Brabrand & Thomsen [38])**.** *Given unambiguous regexes R_1 and R_2:*

(a) $R_1|R_2$ is unambiguous iff $L(R_1) \cap L(R_2) = \phi$.
(b) $R_1 \cdot R_2$ is unambiguous iff $L(R_1) \bowtie L(R_2) = \phi$.
(c) R_1* is unambiguous iff $\epsilon \notin L(R_1) \wedge L(R_1) \bowtie L(R_1*) = \phi$.

In their implementation of this Theorem, Brabrand & Thomsen use Møller's BRICS library [73], and actually rely on what we call the Møller overlap operator, Ω. We use this operator in our theorems. The Møller overlap operator describes only the ambiguous core "a":

$$X\,\Omega\,Y = \{\exists\, x, y \in \Sigma^* \wedge a \mid a \in \Sigma^+ s.t.\ x, xa \in X \wedge ay, y \in Y\}$$

Limitation: Given unambiguous regex components, Theorem 0 specifies when a composed regex remains unambiguous. Yet not all ambiguity is harmful. For example, the regex `\w|\d` is finitely ambiguous. This regex formulation may improve readability [74]; it is not a ReDoS risk.

Regex Infinite Ambiguity Theorems: This section presents our regex ambiguity theory for composition with alternation (Theorem 1), concatenation (Theorem 2), and star (Theorem 3). Here we give proof sketches, examples, and the ReDoS implications. Full proofs are in §B.

Theorem 1 (Ambiguity of Alternation)**.** *Given unambiguous regexes R_1 and R_2,*

(a) $R_1|R_2$ is finitely ambiguous iff $L(R_1) \cap L(R_2) \neq \phi$.
(b) $R_1|R_2$ cannot be infinitely ambiguous.

Proof sketch: The theorem states that given unambiguous regexes R_1 and R_2, if $R_1|R_2$ is ambiguous, then it is always finitely ambiguous. Since R_1 and R_2 are both unambiguous, for any matching input w, there is only one path through R_1 and R_2. Therefore, for $R_1|R_2$ and any matching input w, there are at most two matching paths.

Example: For regex a*|a*, consider input "*a...a*" of length N. Regardless of input length, the number of accepting paths will be 2: via the first $a*$ or the second $a*$.

ReDoS implications: If two regexes R_1 and R_2 are unambiguous, $R_1|R_2$ is always safe (cannot form IA).

Theorem 2 (Ambiguity of Concatenation)**.** *Suppose unambiguous regexes R_1 and R_2, and that $L(R_1) \bowtie L(R_2) = \phi$ (so $R_1{\cdot}R_2$ is ambiguous by Theorem 0). Then:*

(a) $R_1{\cdot}R_2$ is infinitely ambiguous iff $L(R_1)$ contains the language of a regex BC⋆D *and $L(R_2)$ contains the language of a regex* EF⋆G*, where $\epsilon \notin L(\mathtt{C}) \wedge \epsilon \notin L(\mathtt{F}) \wedge L(\mathtt{C}) \cap L(\mathtt{F}) \cap L(\mathtt{DE}) = \phi$.*

(b) Otherwise, $R_1{\cdot}R_2$ must be finitely ambiguous.

Proof sketch: $\Longleftarrow$: Consider the string "*bcc...cdeff...fg*" $\in L(R_1 \cdot R_2)$ where $c = f = de$. It can be divided into two strings "*bcc...cd*" $\in L(\mathtt{BC{\star}D}) \subseteq L(R_1)$ and "*eff...fg*" $\in L(\mathtt{EF{\star}G}) \subseteq L(R_2)$. By hypothesis, we can repeat the substring "*de*" arbitrarily many times, and the resulting string can be matched in R_1 (by C⋆) or in R_2 (by F⋆). We can choose an arbitrarily long string and obtain arbitrary ambiguity in $R_1{\cdot}R_2$.

$\Longrightarrow$: Suppose $R_1{\cdot}R_2$ is infinitely ambiguous. The NFA corresponding to $R_1{\cdot}R_2$ cannot contain the EDA structure because this requires a self-loop — *i.e.,* that R_1 or R_2 is already ambiguous. Therefore the NFA of $R_1{\cdot}R_2$ must contain a PDA structure, as shown in Figure 1(a). We can map the two loops π_1 and π_3 with C⋆ and F⋆ respectively; and the bridge π_2 with DE in the regex representation, where $L(\mathtt{C}) \cap L(\mathtt{F}) \cap L(\mathtt{DE}) = \phi$.

Example: For regex (a⋆a)(aa⋆) on input "*aa...a*" of length N, there are N accepting computations, one for each of the indices of the input dividing the string into a left half consumed by R_1 and a right half consumed by R_2.

ReDoS implications: Though two regexes R_1 and R_2 are unambiguous, $R_1{\cdot}R_2$ could be IA, thus concatenation should be used with care. Theorem 2(a) implies that for $R_1{\cdot}R_2$ to be IA, there must be a star component in both R_1 and R_2. In §5, we introduce three forms of concatenation anti-patterns based on this observation.

Theorem 3 (Ambiguity of Star)**.** *Given unambiguous regex R,*

(a) $R*$ is infinitely ambiguous iff $\epsilon \in L(R) \vee L(R)\ \Omega\ L(R*) = \phi$.

(b) $R*$ cannot be finitely ambiguous.

Proof sketch: The theorem states that given an unambiguous regex R, if $R*$ is ambiguous, then it is always infinitely ambiguous. Suppose $R*$ is ambiguous. Then there is some input w that it can match in k ways, $k > 1$. So there is an input ww that it can match in $k * k = k^2$ ways. The degree of ambiguity increases as a function of input length.

Example: For the regex (a⋆)⋆, consider input "*aaa...a*" of length N. There are two ways (inner ⋆ or outer ⋆) to match each '*a*', making the total number of ways to match to be 2^N.

ReDoS implications: Even though an original regex R is unambiguous, $R*$ can be IA. In §5, we give an anti-pattern that only checks for a subset of conditions for simplicity.

Theorem 4. *Given a finitely ambiguous regex R, $R*$ is always infinitely ambiguous.*

Proof sketch: The proof follows the logic of Theorem 3.

Example: For the regex (a|a)⋆, consider an input "*aaa...a*" of length N. There are two ways (first a or second a) to match each '*a*' of the input, for 2^N matches in all.

ReDoS implications: If R is finitely ambiguous, from alternation ($R = P|Q$) or concatenation ($R = P{\cdot}Q$), $R*$ is always IA. Later in §5, we introduce two anti-patterns of the form $(P|Q)*$.

5. Anti-patterns for Regex Infinite Ambiguity

This section describes anti-patterns for IA regexes (*IA anti-patterns*), derived from the preceding theory of regex infinite ambiguity. Ideal anti-patterns would be as sound and complete as the theory, but this goal must be balanced against usability. With this in mind, we iteratively extracted IA anti-patterns from the theory by dropping clauses from theorems or combining the theorems in different ways. These anti-patterns were refined through internal discussion. We evaluate these anti-patterns in §7 and §8.

Table 1 summarizes our IA anti-patterns. As alternation alone does not make a regex IA (Theorem 1), there are *Concatenation* anti-patterns derived from Theorem 2, and *Star* anti-patterns derived from Theorems 3 and 4.

Concatenation Anti-patterns: The Concat anti-patterns come from Theorem 2. In Theorem 2, a regex R concatenates regexes R_1 and R_2 that contain the languages BC⋆D and EF⋆G, respectively. The theorem states that the potential vulnerability occurs in the sub-regex C⋆DEF⋆, which we write in simplified form as P⋆SQ⋆ for our anti-patterns. We call S the "bridge" between P⋆ and Q⋆.

Concat-1. This anti-pattern, where $P*Q*$ is a sub-regex of R, represents the simplest form without the bridge S. Developers must find a string matched in both $P*$ and $Q*$.

Concat-2. This anti-pattern, where $P*SQ*$ is a sub-regex of R, has the bridge S component. Developers must find a string matched in all $P*$, $Q*$, and S.

Table 1: Our proposed IA anti-patterns. Each row indicates the anti-pattern, the theorem(s) from which it was derived, a description, and an example of how the anti-pattern leads to ambiguity.

Anti-pattern	Thm.	Description	Example
Concat 1	2	`R = ...P⋆Q⋆...` (`R` has a sub-regex `P⋆Q⋆`) — The two quantified parts `P⋆` and `Q⋆` can match some shared string *s*.	`\w⋆\d⋆` — both classes can match digits [0-9].
Concat 2	2	`R = ...P⋆SQ⋆...` — The two quantified parts `P⋆` and `Q⋆` can match a string *s* from the middle part *S*.	`\w⋆0\d⋆` — the repeated classes `\w` and `\d` can match the middle part 0.
Concat 3	2	`R = ...P⋆S⋆Q⋆...` — Advanced form of Concat 1. Since `S⋆` includes an empty string, the ambiguity between `P⋆` and `Q⋆` can be realized.	`\w⋆:⋆\d⋆` — The classes `\w` and `\d` overlap, and the intervening `:⋆` can be skipped.
Star 1	1, 4	`R⋆`, `R= (P|Q|...)` — There is an intersection between any two alternates, *i.e.,* both match some shared strings.	`(\w|\d)⋆` — both classes match digits [0-9].
Star 2	3	`R⋆`, `R= (P|Q|...)` — You can make one option of the alternation by repeating another option multiple times or by concatenating two or more options multiple times.	`(a|b|ab)⋆` — The 3rd option, *ab*, matches combinations of the first and second options.
Star 3	3	`R⋆`, `R= (...P⋆...)` — Nested quantifiers, provided `RR` follows any of the Concat anti-patterns.	Expanding `R=(0?\w⋆)⋆` to `RR` yields `0?\w⋆0?\w⋆`, which is IA by Concat 3. Similarly, `R=(xy⋆)⋆` yields `xy⋆xy⋆`; this is not IA by any Concat anti-pattern.

Concat-3. This anti-pattern, where $P * S * Q*$ is a sub-regex of R, is the case with optional bridge S. Like Concat-1, developers must find a string matched in both $P*$ and $Q*$.

Gap Analysis: The Concat anti-patterns represent all possible ways that the bridge component DE (from Theorem 2) may appear as a sub-regex of the form ϵ, S, or $S*$. Thus, there is no gap between theory and anti-patterns.

Star Anti-patterns: The Star anti-patterns come from Theorem 3 and Theorem 4.

Star-1 and Star-2. These anti-patterns are designed to prevent (some) regexes of the form $R*$ where $R = (P|Q|...)$. Theorem 4 states that if R is finitely ambiguous, then $R*$ becomes IA. From Theorem 1(a), alternations may introduce finite ambiguity. The Star-1 anti-pattern describe the condition when the subregex $(P|Q|...)$ becomes finitely ambiguous. The Star-2 anti-pattern describe the condition when the non-ambiguous $(P|Q|...)$ form IA with the help $*$ according to Theorem 3.

Gap Analysis: There is a gap between Theorem 4 and the Star-1 anti-pattern. Star-1 does not consider all possible forms of finitely ambiguous regexes. For instance, the concatenation may also introduce finite ambiguity (Theorem 2(b)). Thus, some regexes of the form $(P{\cdot}Q)*$ could be IA as well: *e.g.,* `((a|ab)(c|bc))⋆`. Also, the Star-2 anti-pattern is one of the conditions that incorporate Theorem 3. Thus regexes under missing conditions would appear as false negatives for these anti-patterns.

Star-3. This anti-pattern prevent (some) regexes of the form $R*$ where R has a sub-regex $P*$. Theorem 3 states the conditions when $R*$ becomes IA. considering the first condition $\epsilon \in L(R)$ is relatively trivial. Yet, the second condition $L(R)\Omega L(R*) = \phi$ requires reasoning about a language overlap between $L(R)$ and an arbitrary repetition of $L(R*)$, which could be tricky. Based on the common knowledge that a nested quantifier (*e.g.,* $(P*)*$) is bad [75], the Star-3 anti-pattern only considers the case where R has a sub-regex $P*$, as a generalized form of nested quantifiers. The Star-3 anti-pattern further simplifies the condition and asks developers to consider the overlap between $L(R)$ and (twice-repeated) $L(R{\cdot}R)$, using the Concat anti-patterns.

Gap Analysis: The Star-3 anti-pattern does not incorporate all the conditions in Theorem 3. Regexes with the missing conditions would appear as false negatives.

6. Fix Strategies for Regex Infinite Ambiguity

This section describes five fix strategies (F1–F5) that can be broadly applied across the different IA anti-patterns. The fix strategies are derived from various ways of invalidating necessary conditions of Theorem 2 and Theorem 3. The proposed fix strategies do not always preserve semantics.

Fix strategies: Table 2 summarizes the proposed five fix strategies along with examples for each anti-pattern. We evaluate their effectiveness in Experiment 3 (§9).

1) The first fix strategy (F1) is to add a delimiter between the subregexes `P` and `Q` of the anti-patterns (*e.g.,* `P⋆Q⋆`, `(P|Q)⋆`) that can match the shared string(s). More precisely, the delimiter makes $L(C) \cap L(DE) = \phi$ and/or $L(F) \cap L(DE) = \phi$ in Theorem 2(a); and $L(R)\ \Omega\ L(R*) = \phi$ in Theorem 3(a). For instance, consider the regex `\w⋆\d⋆` (Concat 1). If we add a delimiter ':', the new regex `\w⋆:\d⋆` becomes non-IA because $L(\texttt{\w}) \cap L(\texttt{:}) = \phi$ and $L(\texttt{\d}) \cap L(\texttt{:}) = \phi$. Table 2 provides examples for the other anti-patterns.
2) The second fix strategy (F2) is to reduce one of the subregexes `P` and `Q` so that it no longer matches any of the shared strings. In other words, the fix F2 makes $L(C) \cap L(F) = \phi$ in Theorem 2(a); and $L(R)\ \Omega\ L(R*) = \phi$

Table 2: Fix strategies. Each strategy is illustrated with respect to each anti-pattern, within the limit of the example provided.

Fix	**Description**	**Concat 1**	**Concat 2**	**Concat 3**	**Star 1**	**Star 2**	**Star 3**	**Freq.**
		Anti-pattern: ...P*Q*...	...P*SQ*...	...P*S*Q*...	(P|Q|...)*	(P|Q|...)*	(...P*...)*	
		Example: \w*\d*	\w*0\d\*	\w*:*\d*	(\w|\d)*	(a|b|ab)*	(0?\w*)*	
F1	Add a delimiter between the sub-regexes P and Q that can match a shared string.	\w*:\d\*	\w*:0\d\*	\w*:+\d*	(\w*|:\d)\*	(a:|b|ab)*	(:0?\w*)*	12
F2	Reduce one of the sub-regexes P and Q so that it no longer matches any of the shared strings.	[a-zA-Z_]*\d*	[a-zA-Z_]*0\d*	[a-zA-Z_]*:*\d*	([a-zA-Z_]|\d)*	(b|ab)*	(0?[a-zA-Z_])*	10
F3	Reduce both the sub-regexes P and Q so that they no longer match any of the shared strings, and add the shared string(s) in a disjunction. In many cases, this will resemble making a superset of the two sub-regexes.	\w*	N/A	N/A	\w*	(a|b)*	\w*	3
F4	Reduce or remove repetition in at least one of the sub-regexes P and Q that match a shared string.	\w{,10}\d{,10}	\w{,10}0\d{,10}	\w{,10}:*\d{,10}	(\w|\d){,10}	(a|b|ab){,10}	(0?\w{,10}){,10}	13
F5	Remove or substantially modify the sub-regexes P and Q and add logic for the semantic changes.	add logic to catch non-digits then use \d*	add logic to catch non-digits then use 0\d\*	add logic to catch non-digits then use :*\d*	\w*|\d*	a+|b+|(ab)+	add logic to catch 0, then use \w*	16

in Theorem 3(a). For instance, refer to the same regex `\w*\d*` (Concat 1). If we reduce `\w` to `[A-Za-z_]` so that it does not overlap with `\d`, the new regex `[A-Za-z_]*\d*` is not IA since $L(\texttt{[A-Za-z\_]}) \cap L(\texttt{\textbackslash d}) = \phi$.

3) The third fix strategy (F3) is to reduce the subregexes `P` and `Q` so that they no longer match any of the shared strings, and add the shared string(s) in a disjunction. In many cases, this will resemble making a superset of the two sub-regexes. Effectively, the fix F3 has the same effect as F2 that excludes any shared string(s), yet it additionally keeps the shared string(s) in a disjunction, making the fix semantic-preserving. For example, consider `(\w|\d)*` (Star 1). Suppose we reduce `\w` to `[A-Za-z_]` and reduce `\d` to null so that they no longer match the shared string(s) `[0-9]`. Then we add the subregex `[0-9]` in a disjunction. Finally, we get `([A-Za-z_]|[0-9])*`, which is equivalent to `\w*`. Note that `\w` is a superset of `\w` and `\d`, and the old and new regexes match the same language.
4) The fourth fix strategy (F4) is to reduce or remove repetition in at least one of the subregexes P and Q so that the shared string(s) cannot be matched infinitely. Both Theorem 2 and Theorem 3 require an unbounded repetitions (a star quantifier). The fix F4 in effect turns unbounded repetitions to bounded ones. For example in Star 1 anti-pattern, we can replace the regex `(\w|\d)*` with `(\w|\d){0,10}` permitting only up to 10 repetitions.
5) The fifth fix strategy (F5) is to remove or substantially modify the subregexes P and Q and handle semantic changes elsewhere. The fix F5 capture general non-systematic fixes that may introduce more semantic changes than the other fixes. For example, the regex `(\w|\d)*` can be fixed to `\w*|\d*`.

Evaluation of Practical Relevance: We analyzed the 54 developer-created regex fixes reported by Davis *et al.* [2]. We classified each fix into one of these five strategies. The last column in Table 2 reports the frequency of each fix strategy. We also observed that developers value simplicity in the fix. To preserve the original (vulnerable) regex's structure, they introduced semantic changes (51/54 = 94%).

7. Experiment 1: Effectiveness of Anti-patterns

Our proposed IA anti-patterns (§5) were derived from our theory (§4), but we deliberately introduced inaccuracy in favor of simplicity. In this section, we evaluate the impact of these deviations over the largest available regex corpus [32]. We measured effectiveness using precision and recall.

7.1. Experimental Design

Studied Techniques: *Our IA anti-patterns.* We detected each anti-pattern using static analysis. We parsed regexes in PCRE format [36] using an ANTLR 4 grammar and parser [76]. We used the BRICS [73] tool to check whether multiple sub-regex parts can generate any shared string.

We implemented our IA anti-patterns to support the common case of K-regexes, *i.e.,* regexes that use only Kleene-regular regexes (cf. §2). Our prototype also excludes extended POSIX and Unicode character classes for simplicity. These limitations are consistent with past approaches [17]–[20], [22]–[25].

State-of-the-art (SOA) anti-patterns. For comparison, we also executed the state-of-the-art (SOA) anti-patterns that characterize IA regexes [2]. We used the automatic detector provided by Davis *et al.* [2]. These anti-patterns lack a

Table 3: State of the art IA anti-patterns, as described by Davis *et al.* [2]. Each row indicates the anti-pattern, its description. and an example.

Anti-pattern	Description
QOA (Quantified Overlapping Adjacency) Example: `/\w*#?\w*/`	*The two quantified* `\w*` *nodes overlap, and are adjacent because one can be reached from the other by skipping the optional octothorpe. 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).*
QOD (Quantified Overlapping Disjunction) Example: `/(\w|\d)+/`	*Here we have a quantified disjunction* `(/(...|...)+/)`*, whose two nodes overlap in the digits, 0-9.*
Star height >1 Example: `/(a+)+/`	*To measure star height, we traverse the regex and maintain a counter for each layer of nested quantifier: +, *, and check if the counter reached a value higher than 1. In such cases, 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+)+/`*.*

theoretical basis, so we expect them to perform worse. Davis *et al.* described three anti-patterns, listed in Table 3[2]

Ground Truth: We assessed ground truth for whether a regex is IA using Weideman *et al.*'s detector [18], [19]. This detector tests if a regex is IA by analyzing its NFA (Figure 1). Since the Weideman tool uses automata theory instead of regex semantic theory, it provides an independent check on the anti-patterns (and our underlying theory).

Metrics: The standard metrics of precision and recall [77].

Dataset: We evaluated the studied anti-patterns in the largest available dataset of real-world regexes [32] (537,806 regexes). This dataset has been used by previous studies for measuring ReDoS [32], fixing ReDoS [43], [78], and measuring general characteristics of regexes [37]. We analyzed 209,188 regexes from this dataset — one order of magnitude larger than the evaluation of past ReDoS-detection approaches (15,000–30,000 regexes [18], [19], [21], [26]).

We curated the dataset for this experiment: (1) We removed the 295,151 regexes that were not supported by the ground truth tool [19]).[3] According to our ground truth, 32,005 of our studied regexes were IA. (2) We discarded 32,413 additional regexes that were not supported by the BRICS library [73] used in our anti-pattern prototype. (3) We discarded 1,054 additional regexes with POSIX or Unicode character classes not supported by our implementation.

Table 4: Comparison of Precision and Recall between our IA anti-patterns and SOA anti-patterns.

Anti-patterns	Precision	Recall
Our IA anti-patterns	100%	99%
SOA anti-patterns [2]	50%	87%

The implementation of our IA anti-patterns supports 450,753 regexes (83.8%) of the dataset (10.2% had advanced or non-regular features; 6% unsupported by BRICS). This level of completeness is comparable to prior research prototypes for regex analysis [20], [23], [43].

Finally, we measured regex generalizability metrics [37] in the regexes that we kept and filtered out. We found that they were similar in median: length (18 vs. 19), paths (1 vs. 1), features (3 vs. 4), and ratio of IA regexes flagged by our anti-patterns (15% vs. 15.8%).

7.2. Results

How Effective were Our IA Anti-patterns Compared to the SOA Anti-patterns?. In Table 4, we report the results for our studied anti-pattern families. It shows that our proposed IA anti-patterns provided a substantial improvement in both precision (100% compared to 50%) and recall (99% compared to 87%) when compared to the SOA anti-patterns. Our IA anti-patterns addressed many of the false positives of the SOA. For example, the *Star height* > 1 anti-pattern can produce many false positives *e.g.,* the non-IA regex `/(b*c)*/` has *Star height = 2*. Our IA anti-patterns also reduced the number of false negatives of the SOA, *e.g.,* for regexes like `(a|b)*(ab)*` and `(a|b|ab)*`. The SOA anti-patterns find no overlap between $(a|b)$ and (ab) and would not label them as IA. In contrast, our *Concat 1* and *Star 2* anti-patterns, respectively, would label both as IA.

We note that we observed higher precision and recall achieved by the SOA anti-patterns than was reported by their original study [2]. We suggest two reasons: we studied a different dataset, and we assumed full match for unanchored regexes (*e.g.,* converting `a+` to `/^.*?a+$/`) [32]), which reveals more IA regexes in the dataset.

Finally, we also performed a deeper investigation into the root cause of the false negatives of our IA anti-patterns (the 1% of IA regexes that they did not flag as IA). The false negatives in our experiment were mainly regexes with constructions that were too complex for our current anti-pattern scripts to detect for the limitation of Star anti-patterns discussed in §5. While this limitation of our implementation caused a few false negatives (affecting only 1% of IA regexes), our implementation is still sound for our studied dataset — it caused no false positives.

How Prevalent was each of our IA Anti-patterns?. Table 5 shows the prevalence of each of our proposed IA anti-patterns in our studied dataset, *i.e.,* the ratio of IA regexes that were detected by each IA anti-pattern. Note that the

[2] As might be expected, these anti-patterns resemble those presented in Table 1. The main difference is in the nuanced definition of "overlap" available from our IA theory.

[3] While our implementation of our IA anti-patterns supported a larger percentage of the dataset, we discarded those for which we could not collect ground truth.

Table 5: Prevalence of each of our proposed IA anti-patterns within the studied dataset. As some regexes fit multiple IA anti-patterns, the final row eliminates double-counting.

IA Anti-pattern	# Regexes	Prevalence
Concat 1	17,349	54%
Concat 2	12,419	39%
Concat 3	414	1%
Star 1	192	<1%
Star 2	639	2%
Star 3	1,133	4%
All anti-patterns	31,537	99%

prevalence ratios do not add up to 100%, since some IA regexes may contain multiple anti-patterns.

We make multiple observations in this table. First, all our IA anti-patterns as a group provided high recall (99%); false negatives are rare. Second, we observed wide variations in the prevalence of each individual anti-patterns. This means that our set of anti-patterns could be further simplified and still obtain very high recall altogether. Somebody wanting to learn only a single anti-pattern could learn only Concat 1 and still cover 54% of IA regexes — adding Concat 2, one would cover the large majority ($> 90\%$) of IA regexes, and so on. This confirms past research that found that polynomial regexes were much more prevalent than exponential ones [62]. We believe that this does not mean that developers are already good at avoiding some anti-patterns, but instead that the kinds of problems that would require a Concat regex are more common than those that would require a Star one. However, future work would be needed to answer this question. Finally, before considering ignoring the less common (lower prevalence) IA anti-patterns, one should also consider their risk. While the star anti-patterns are less common (about 6% of all IA regexes), they are riskier — our theory shows that they lead to exponential ambiguity.

Summary for Experiment 1: Our IA anti-patterns correctly identified IA regexes with substantially higher effectiveness (100% precision, 99% recall) than the SOA anti-patterns (50% precision, 87% recall).

8. Experiment 2: Effectiveness when Applied by Humans

Our IA anti-patterns can identify IA regexes with high precision and recall (§7), but their effectiveness may be reduced when applied manually [31]. Here we report on a human-subjects experiment evaluating the effectiveness of our IA anti-patterns when applied manually.

8.1. Experimental Design

Overview: We asked 20 software developers to perform 5 regex composition tasks. To study a context in which developers may prefer to apply IA anti-patterns manually [31], we studied simple regex composition tasks. We followed a within-subjects approach: each subject applied both our IA anti-patterns and the state-of-the-art (SOA) ones. Among 20 participants, half (10) used our anti-patterns first, and the other half used the SOA ones first. We measured whether subjects correctly identified IA in their regexes.

Treatments: We showed subjects our IA anti-patterns as described in Table 1 and the SOA anti-patterns using verbatim text from Davis *et al.*'s original description of the anti-pattern, and of how it should be applied (described in Table 3). Note that we did not study a control group that used no anti-patterns. Experiment 1 already answers what a control group would show: when developers are given no support, they write thousands of vulnerable regexes (§7).

Tasks: Table 9 shows the 5 tasks of Experiment 2. Task 1 was an easy warm-up task, to familiarize subjects with the structure of the experiment. The next three tasks (Tasks 2, 3, 4) evaluated limitations that we identified in the three SOA anti-patterns, to learn if our IA anti-patterns were more effective in those scenarios. In task 2, *Star height* > 1 may produce a false positive, assessing the regex as IA. In task 3, *QOA* may produce a false negative, assessing the regex as non-IA. In task 4, *QOD* may produce a false negative. Finally, task 5 evaluated a scenario in which the SOA anti-patterns are successful, to learn if our IA anti-patterns are comparable in such a scenario. We expected both sets of anti-patterns to perform equally in tasks 1 and 5, and our IA anti-patterns to be more effective in tasks 2, 3, and 4.

Within-Subjects Protocol: Our protocol had three steps: (1) *Training:* We shared background information in: (*i*) regex syntax and useful terminology, so that they knew correct regex syntax; and (*ii*) regex ambiguity and ReDoS, so that they understood the practical utility of the task and thus increase their engagement. (2) *First set of anti-patterns:* We taught subjects one set of anti-patterns. They completed the 5 regex composition tasks, producing a regex that is not IA, using the given anti-patterns. (3) *Second set of anti-patterns:* We taught subjects the other set of anti-patterns. They performed the same 5 tasks, in the same order, using the other anti-patterns.

We let subjects ask clarifying questions. We asked them to think aloud. The experiment took $\sim$1 hour per subject. Subjects were compensated with a $15 gift card.

Subjects: Subjects were recruited via posts on Twitter, Reddit (r/regex), and our institutional mailing lists. We asked subjects to report their years of professional software development and their experience with regexes (self-reported, based on popular regex features following Michael *et al.* [46]). We had 27 respondents, and kept the 21 respondents who reported some experience in both categories. After performing the experiment, we discarded one additional subject who composed incorrect regexes for 70% of the tasks (they did not match the example inputs provided in the specification). Thus, in total, we analyzed the performance of 20 subjects. We list their demographics in Table 7.

Metrics: For each studied task and anti-pattern set, we measured success using *Detection Effectiveness*: the percentage of subjects that correctly identified whether their composed

Table 6: Regex composition tasks studied in Experiment 2.

Task	Description	Typical solution
1	Write a regex to match one or more 'b' followed by a single 'c'. Example matching strings: bc, bbc, bbbbc, bbbbbc, bbbbbbbbbbbbbbbbbbbbbc	non-IA: `b+c`
2	Write a regex to match one or more repetitions of the following: one or more 'b' followed by a single 'c'. Example matching strings: bcbc, bbcbbcbbc, bbbbbcbbbbbc, bbbbbbbbbbbbbbbbbbbbbcbbbbbbbbbbbbbbbbbbbbbc	non-IA: `(b+c)+`
3	Write a regex to match one or more 'a' or 'b', followed by one or more repetitions of 'ab'. Example matching strings: aab, bab, aaab, aaaaab, bab, bbbab, aaaabababab, bbbbabababababab	IA: `(a|b)+(ab)+`
4	Write a regex to match one or more occurrences of the strings 'a', 'b', or 'ab'. Example matching strings: aaaaaaaaaa, bbbbbbbbbbb, abababababababab	IA: `(a|b|ab)+`
5	Write a regex to match one or more 'a' followed by an optional 'b' followed by one or more 'a'. Example matching strings: aaaabaa, aaaaa, abaaaa	IA: `(a+b?a+)`

Table 7: Demographics of Experiment 2: subjects' experience with software development, and with regexes.

	Years Prof. Soft. Dev.			**Exp. with Regexes**		
	< 1	1-2	3-5	*Novice*	*Interm.*	*Expert*
# Subjects	7	7	6	9	11	0

regex was IA. We used the same approach for ground truth as in Experiment 1 (§7.1). Note that we did not measure whether subjects fixed the IA section in their regex, if any; we measured the effectiveness of anti-patterns following the goal of the original SOA anti-patterns — to identify IA.

Statistical Tests: We validated our results using hypothesis testing and power analysis.

Hypothesis testing. We used the null hypothesis: H_0*: subjects using our IA anti-patterns achieve as much detection effectiveness as those using the SOA anti-patterns.* We tested H_0 using a Wilcoxon signed rank test [79] (since we cannot assume a normal distribution of results, and our observations are paired) over the IA assessments produced by our IA anti-patterns and the SOA ones for all tasks and orders. If this test returned a low p-value ($p < 0.05$), we rejected H_0.

Power analysis. We used power analysis to determine if our sample size was sufficient to support a statistically significant expected effect size in detection effectiveness [80]. We looked for standard power of 0.8, standard statistical significance of $p < 0.05$, with our observed effect size (*i.e.,* the difference in detection effectiveness using our IA anti-patterns vs. using the SOA anti-patterns for all tasks and orders).

Table 8: Performance of subjects in Experiment 2: percentage of subjects correctly using each anti-pattern set to identify if their composed regex was IA.

	SOA first, IA after ($N = 10$)		**IA first, SOA after** ($N = 10$)		**All orders** ($N = 20$)	
Task	**SOA**	**IA**	**SOA**	**IA**	**SOA**	**IA**
1	100%	100%	100%	100%	100%	100%
2	10%	100%	0%	100%	5%	100%
3	20%	100%	20%	100%	20%	100%
4	30%	100%	20%	100%	25%	100%
5	100%	100%	100%	100%	100%	100%
All	52%	100%	48%	100%	50%	100%

8.2. Results

Table 8 summarizes our results for each order of application, by each set of anti-patterns, for each task.

- Considering all tasks and treatment orders, subjects using our IA anti-patterns achieved 100% detection effectiveness, improving on the SOA anti-patterns (50%).
- Our hypothesis test showed a statistically significant improvement ($p < .00001$).
- Our observed effect size was 50%, comparing the final columns in the bottom row of Table 8. Our power analysis indicated that we studied a sufficient number of subjects: we needed 11 and studied 20.

As we expected (§8.1), the SOA anti-patterns showed their limitations in tasks 2, 3, and 4 — regardless of the ordering. Also as expected, both sets of anti-patterns achieved 100% detection effectiveness for tasks 1 and 5, also regardless of ordering. We conclude that our IA anti-patterns are as effective as the SOA ones when they are not limited, and much more effective than them when they are.

Summary for Experiment 2: Our IA anti-patterns outperformed the SOA anti-patterns when applied manually (100% vs. 50% effectiveness).

9. Experiment 3: Usability when Complementing Existing Tools

Experiments 1 and 2 showed that our anti-patterns are effective over a wide variety of regexes (§7), and can be applied manually by humans (§8). However, developers may prefer automatic tools, *e.g.,* for complex regexes.

In Experiment 3, we studied whether our anti-patterns and fix strategies *complement* automatic tools for real-world regexes. Our goal is not to replace existing automatic tools (we hope developers use them!), but to complement them, to increase developer understanding of the task outcome.

9.1. Experimental Design

Overview: We asked 9 software developers to perform real-world ReDoS detection and fixing. They performed tasks over their own regexes from open-source projects. They first used only an automatic tool, and then the tool combined with our anti-patterns (for detection) and our fix strategies (for fixing). Our design was within-subjects. We fixed the order so we could measure their (hypothesized) increase in understanding after adding our approach.

Treatments: For *detection*, our subjects first used a representative detection tool (Weideman *et al.*'s [19]), and then complemented it with our anti-patterns. We studied Weideman *et al.*'s approach because they provide a mature implementation with many stars in GitHub. For *fixing*, our subjects first used a representative fixing approach (van der Merwe *et al.*'s [27]), and then complemented it with our anti-patterns and fix strategies. We studied van der Merwe *et al.*'s approach because it is the only existing fixing approach that does not modify the language accepted by the regex.[4]

To help these tools to perform their best, we trained our subjects. For *detection*, we trained them on the purpose and workings of Weideman's detection tool, including the NFA-based characterizations of IA that it detects in the regex (see Figure 1). For *fixing*, we trained them on the purpose and workings of Van Der Merwe's algorithm, *i.e.,* that it converts the regex's NFA to an equivalent unambiguous DFA, then back to an equivalent regex. We also explained our anti-patterns (Table 1) and fix strategies (Table 2).

Tasks: Table 9 shows the tasks of Experiment 3.

Detection Task. We asked our subjects to detect vulnerability in 3 regexes, in a random order: a PDA regex, an EDA regex, and a non-IA regex (see Figure 1). One of the PDA or EDA regexes was the vulnerable one that we took from the subject's software project. For the other two regexes, we used the same randomly chosen regexes from the dataset studied in Experiment 1 (§7.1). For each regex, we showed our subjects the output of Weideman's detection tool, and asked them how strongly they understood the vulnerability in the regex. Then, we also showed them our anti-patterns, asked them to identify the anti-pattern(s) that each regex fits (to prompt them to use the anti-patterns), and asked them the same question again.

Fixing Task. We asked our subjects to fix the vulnerability in the regex that we took from their code. We showed them their regex in the context of their project, and asked them how strongly they understand it (to refresh their memory). Then, we showed them the output of van der Merwe's approach: a non-IA version of their regex. We let them write their own fix or take/adapt van der Merwe's, and we asked them our understanding questions. Then, we also showed them our anti-patterns and fix strategies, again let them modify their fix if they choose to, and again asked them our understanding questions.

Within-Subjects Protocol: Our protocol had three steps: (1) *Training:* This training was more in-depth than Experiment 2 because the subjects had greater expertise. We taught the technical details of ReDoS attacks (following [2]), and showed the participants how the detection [19] and fixing [27] tools work. (2) *Detection:* We asked subjects to detect IA in a set of regexes, first using only existing automatic tools, and then combining them with our anti-patterns. (3) *Fixing:* We asked subjects to fix an IA regex that they wrote in their codebase to make it non-IA, first using only existing automatic tools, and then combining them with our anti-patterns and fix strategies.

Table 9: Tasks of Experiment 3. Italics denote changing text with each subject. Brackets denote subject answers.

Detection Task

	Output of automatic detection tool	**How strongly do you understand what makes this regex vulnerable?**	**Explain your reasoning**
PDA regex	*Output of Weideman's detection tool* [19]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[. . .]
EDA regex	*Output of Weideman's detection tool* [19]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[. . .]
Non-IA regex	*Output of Weideman's detection tool* [19]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[. . .]

Fixing Task

	Output of automatic fixing tool	**How strongly do you understand what makes the resulting fixed regex not vulnerable?**	**Explain your reasoning**
Their vulnerable regex in context	*Output of van der Merwe's fixing tool* [27]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[. . .]

We let subjects ask clarifying questions. We asked them to think aloud. To simulate real-world conditions, we let them use external resources, and showed them some resources: a web interface for the studied tools, and two websites for regex understanding (www.regex101.com and www.regexper.com). The experiment took ∼1 hour per subject. Subjects were compensated with a $40 gift card.

Subjects: We recruited software developers that had written a vulnerable regular expression in the PyPi [81] and NPM [82] software ecosystem. To increase response rate, we scanned ∼200K PyPi projects and ∼40K NPM projects with some popularity (at least 1 star) and with recent activity (at least one commit since January 2020). We extracted their regexes; discarded any in test files or dependencies; and identified vulnerable regexes using Davis *et al.'s* ensemble of ReDoS detectors [32], This process resulted in 120 vulnerable regexes (99 from PyPI, 21 from NPM). We disclosed these potential vulnerabilities to the 120 software developers who last modified them. We invited those developers to participate in our experiment. 9 of them agreed (8% response rate) — demographics are in Table 10.

Metrics: We measured the success of our anti-patterns or fix strategies as the increase in understanding that our

[4] The algorithm of van der Merwe *et al.* does not include an open-source implementation. Our implementation is included in our artifact.

Table 10: Demographics of Experiment 3: subjects' experience with software development, and with regexes.

	Years Prof. Soft. Dev.			Exp. with Regexes	
	3-5	6-10	> 10	*Interm.*	*Expert*
# Subjects	1	1	7	1	8

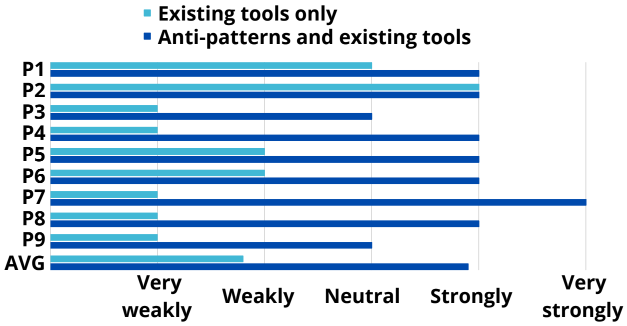


Figure 2: Detection Task: Subjects consistently reported stronger understanding of what makes their regex vulnerable when using our anti-patterns to complement existing tools.

subjects reported after applying them. We asked our subjects the same question twice, once after applying each treatment. We also asked them to explain their reasoning (see Table 9).

For *detection*, we asked them how strongly they understood what makes the regex vulnerable, using a Likert scale of: *"Very strongly"*, *"Strongly"*, *"Neutral"*, *"Weakly"*, and *"Very weakly"* understand, and *"Not vulnerable"*. For *fixing*, we asked them how strongly they understood what makes the resulting fixed regex not vulnerable, using the same scale. Finally, we asked them how helpful they found the anti-patterns or fix strategies for their future overall.

Statistical Tests: We validated our results using hypothesis testing and power analysis (as in §8.1).

Hypothesis testing. We set two null hypotheses. For detection, H_0: *Subjects using our IA anti-patterns in combination with existing tools report the same understanding strength of what makes the regex vulnerable as those using existing tools only.* For fixing, H_0: *Subjects using our fix strategies in combination with existing tools report the same understanding strength of what makes the fixed regex not vulnerable as those using existing tools only.* We tested the null hypothesis for each task using a Wilcoxon signed rank test [79] (since we cannot assume a normal distribution of results, and our observations are paired) over the reported understanding scores for each treatment.

Power analysis. We again looked for standard power of 0.8, standard statistical significance of $p < 0.05$, and measured effect size as the increase in mean reported understanding for each task.

9.2. Results

Detection Task: We focus on how strongly subjects understood what makes their own regex vulnerable (see Figure 2).

- Subjects using our anti-patterns to complement existing tools reported median *"Strongly"* understanding the vulnerability, improving over using existing tools only (median *"Very weakly"*).
- Our hypothesis test showed a statistically significant improvement ($p < 0.05$).
- Our observed effect size was mean 2.1 Likert points — bottom bar in Figure 2. Our power analysis indicated that we studied a sufficient number of subjects: we needed 4 and studied 9.

Subjects also reported that the anti-patterns will be *"Helpful"* ($N = 4$) or *"Very helpful"* ($N = 5$) for their future detection efforts. The following quote describes their most common sentiment: "*I will use the tool to see if there is something wrong, and with the anti-patterns I can try to understand why there is a problem*".

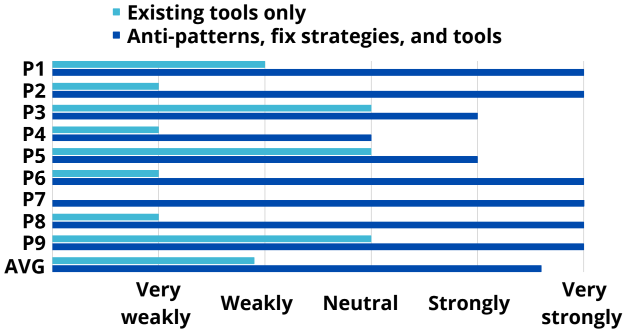


Figure 3: Fixing Task: Subjects consistently reported stronger understanding of what makes their resulting fixed regex not vulnerable when they used our anti-patterns and fix strategies to complement existing tools.

Fixing Task: Figure 3 shows our subjects' reported understanding of what makes the fixed regex not vulnerable.

- Subjects using our anti-patterns and fix strategies to complement existing tools reported median *"Very strongly"* understanding, improving over using existing tools only (median *"Very weakly"*).
- Our hypothesis test showed a statistically significant improvement ($p < 0.05$).
- Our observed effect size was mean 2.9 Likert points — bottom bar in Figure 3. Our power analysis indicated that we studied a sufficient number of subjects: we needed 2 and studied 9.

Subjects also reported that the anti-patterns and fix strategies will be *"Neutral"* ($N = 1$), *"Helpful"* ($N = 2$), or *"Very helpful"* ($N = 6$) for their future detection efforts. As an example quote, one subject regarded the fixing tool as: "*The output does not make a whole lot of sense to me*". Another said of the fix resulting from our fix strategies: "*I understand why this is ambiguous and how the change fixes it*". Finally, almost all subjects ($N = 8$) were more comfortable fixing their codebase with the fix produced using our fix strategies than with the one produced by the existing tool (Figure 5 in Appendix).

Summary for Experiment 3: Subjects using our IA anti-patterns and fix strategies to complement existing tools reported much higher understanding, from median *"Very weakly"* to median *"Strongly"* for detection, and to median *"Very strongly"* for fixing.

10. Threats to Validity

Internal Validity: We took multiple measures to increase internal validity. In Experiment 1 (§7), we tested the implementation scripts of our anti-patterns over small samples of the dataset. We also curated our studied dataset to prepare it for our experiments. We also used existing implementations of tools where possible, viz. the SOA anti-patterns by Davis *et al.* [2] and Weidemann *et al.*'s detector [19], to avoid errors if implementing them ourselves.

In Experiment 2 (§8), we used best practices in human-subject experiment methodologies in its design, *e.g.,* [83], [84]. We *piloted* the protocol on 3 pilot studies, which helped us clarify the language describing the tasks (pilot 1) and the technical terms in the training (pilot 2). Pilot 3 showed that our script was adequate. To reduce *social desirability bias*, we did not disclose who created any of the anti-patterns, and we referred to them in the third person (Anti-patterns 1 and 2). To avoid *expertise bias*, we asked subjects to apply the given anti-patterns irrespective of their perception of their correctness. To avoid *learning bias*, 10 random subjects used the SOA anti-patterns first, and the other 10 used our IA anti-patterns first.

In Experiment 3 (§9) we likewise used best practices in design. We *piloted* the protocol on 3 subjects. After pilot 1, we adjusted the number of tasks to reduce the experiment duration. After pilots 2 and 3, we clarified some terms in the training. To reduce *social desirability bias*, we did not disclose who created any of the tools or supplementary approaches (even if we were asked); we referred to them only as "Treatment 1" etc. Further, we separately and independently asked subjects their understanding using one treatment and then using the other (as opposed to asking them to compare treatments). Having subjects individually decide and assign a specific score to each treatment reduces the possibility of them unconsciously preferring the last treatment. To reduce *expertise bias* (*e.g.,* higher understanding reported by more experienced subjects), all subjects used both treatments. To reduce *learning bias*, subjects used first the treatment that we anticipated would provide lower understanding, *i.e.,* the existing tools. If subjects used first the treatment that truly provided higher understanding (and second the one that truly produced lower), they would misleadingly report higher understanding for the second treatment; they cannot forget what they learned. Having the combination of our anti-patterns, fix strategies, and existing tools as the last treatment may have unconsciously nudged subjects to report higher understanding for them. However, subjects reported much higher understanding for this last treatment with statistical significance, *i.e.,* more likely due to a real effect than to chance.

External Validity: We also took multiple measures to increase external validity.

In Experiment 1 (§7), we evaluated the largest available dataset of regexes [32].

In Experiment 2 (§8), our subjects had diverse levels of professional software development experience. Their experience with regexes was at the novice and intermediate levels, but we studied regex experts in Experiment 3. We studied simple composition tasks to represent situations when developers may choose to apply anti-patterns manually, but they represented diverse scenarios. Furthermore, our subjects composed solutions that were only slightly less complex than typical real-world regexes according to [37]. For example, they had length 6-11 (median regex length in Java: 15) and used 2-3 operators (Java: median 3). We report the most common solution observed for each task in Table 6. We also studied more complex real-world regexes in Experiment 3.

In Experiment 3 (§9), we studied mostly regex experts and complex real-world regexes to complement Experiment 2. We also made this experiment as realistic as possible by having developers work with their own regexes, and giving them free access to online resources.

Finally, both Experiment 2 and 3 studied a limited number of subjects ($N = 20$ and $N = 9$). However, we observed large effect sizes in our results, we did so consistently, they were statistically significant, and power analysis revealed that fewer subjects would have been sufficient.

11. Conclusions

To secure software systems, developers need approaches that are both sound and understandable. Prior to this paper, the approaches to address regular expression security problems provided theoretical guarantees, but were difficult for developers to understand. Our goal was to complement these existing approaches with understandable regex security anti-patterns and fix strategies. To that end, we developed a novel theory of regex infinite ambiguity that characterizes vulnerable regexes to ReDoS, and a set of anti-patterns and fix strategies derived from it. Our evaluation showed that our IA anti-patterns identified vulnerable regexes with much higher effectiveness than the state-of-the-art anti-patterns, both when applied automatically and manually. Our anti-patterns and fix strategies also substantially increased developer understanding when used alongside existing tools to detect and fix vulnerable regexes. In the future, we plan to apply this methodology to similar security problems in domain-specific languages (*e.g.,* in GraphQL [85]).

Research Ethics

Our human subjects experiments were overseen by the appropriate Institutional Review Board (IRB).

Acknowledgments

We thank the reviewers for their constructive feedback. We thank Charles M. Sale for developing the www.regextools.io platform for our experiment. Lee and Davis acknowledge support from NSF award #2135156, and Servant from URJC award C01INVESDIST.

Appendix A. Replication Package

We have made our data and code publicly available for replication [33].

It contains, for Experiment 1: *i)* the dataset used (§7.1), *ii)* our implementation of Weideman's detection tool [19] used as ground truth (§7.1), and *iii)* the implementation of our anti-patterns (§7.1). For Experiments 2 and 3: *iv)* the full protocol used (§8.1 and §9.1), and *v)* our implementation of van der Merwe's fixing tool [27] (§9.1). Finally, *vi)* the analysis of prevalence of our fix strategies (§6).

Appendix B. Proofs of the Theorems

B.1. Definitions

We define the operators used in Theorems 2 and 3:

B.1.1. $\bowtie$. Brabrand & Thomsen [38] introduced an overlap operator, $\bowtie$, between two languages $L(R_1)$ and $L(R_2)$. The set $L(R_1) \bowtie L(R_2)$ contains the ambiguity-inducing strings that can be parsed in multiple ways across $L(R_1)$ and $L(R_2)$. More formally, with $X = L(R_1)$ and $Y = L(R_2)$,

$$X \bowtie Y = \{xay \mid x, y \in \Sigma^* \wedge a \in \Sigma^+ s.t.\ x, xa \in X \wedge ay, y \in Y\}$$

B.1.2. Ω. Brabrand & Thomsen use Møller's BRICS library [73] for the implementation of their theorems, and actually use what we call the "Møller overlap operator", Ω. We use this operator in our theorems. The Møller overlap operator describes only the ambiguous core "a":

$$X\,\Omega\,Y = \{\exists\, x, y \in \Sigma^* \wedge a \mid a \in \Sigma^+ s.t.\ x, xa \in X \wedge ay, y \in Y\}$$

B.2. Assumptions

In our theorems and proofs, we assume that we can convert regexes to their equivalent, ambiguity-preserving, ϵ-free NFAs [19], [40].

B.3. Theorems & proofs

Brabrand & Thomsen's Theorem 0 [38] provides the conditions for *unambiguity*. Our proofs consider the effect of negating the unambiguity condition, and distinguish the conditions that lead to finite or infinite ambiguity.

B.3.1. Theorem 0: Brabrand & Thomsen's [38] Theorem. *Given unambiguous regexes R_1 and R_2,*

(a) $R_1|R_2$ is unambiguous iff $L(R_1) \cap L(R_2) = \phi$.
(b) $R_1{\cdot}R_2$ is unambiguous iff $L(R_1) \bowtie L(R_2) = \phi$.
(c) $R_1{*}$ is unambiguous iff $\epsilon \notin L(R_1) \wedge L(R_1) \bowtie L(R_1{*}) = \phi$.

B.3.2. Theorem 1: Ambiguity of Alternation. *Given unambiguous regexes R_1 and R_2,*

(a) *$R_1|R_2$ is finitely ambiguous iff $L(R_1) \cap L(R_2) = \phi$.*
(b) *$R_1|R_2$ cannot be infinitely ambiguous.*

The components of Theorem 1 follow from Lemma 1.

Lemma 1. *Given unambiguous R_1 and R_2, if $R_1|R_2$ is ambiguous it is always finitely ambiguous.*

Proof. A string s may be matched against $R_1|R_2$ in four ways: s may be matched by R_1, by R_2, by both, or by neither. In any case, since R_1 and R_2 are unambiguous, there are at most two ways for $R_1|R_2$ to match s. □

B.3.3. Theorem 2: Ambiguity of Concatenation. *Suppose unambiguous regexes R_1 and R_2, and that $L(R_1) \bowtie L(R_2) = \phi$ (so $R_1{\cdot}R_2$ is ambiguous by Theorem 0). Then:*

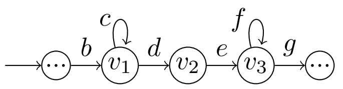
(a) $R_1{\cdot}R_2$ is infinitely ambiguous iff $L(R_1)$ contains the language of a regex $\texttt{BC}{\star}\texttt{D}$ and $L(R_2)$ contains the language of a regex $\texttt{EF}{\star}\texttt{G}$, where $\epsilon \notin L(\texttt{C}) \wedge \epsilon \notin L(\texttt{F}) \wedge L(\texttt{C}) \cap L(\texttt{F}) \cap L(\texttt{DE}) = \phi$.
(b) Otherwise, $R_1{\cdot}R_2$ must be finitely ambiguous.

2(a) is an iff so we need to prove:

$\Longleftarrow$: If $L(R_1)$ contains the language of a regex $\texttt{BC}{\star}\texttt{D}$ and $L(R_2)$ contains the language of a regex $\texttt{EF}{\star}\texttt{G}$, where $\epsilon \notin L(\texttt{C}) \wedge \epsilon \notin L(\texttt{F}) \wedge L(\texttt{C}) \cap L(\texttt{F}) \cap L(\texttt{DE}) = \phi$, then $R_1 \cdot R_2$ is infinitely ambiguous.

Proof. Consider a string $q = bc^m d \in L(\texttt{BC}{\star}\texttt{D})$ where $b, d \in \Sigma^*$, $c \in \Sigma^+$, $b \in L(\texttt{B})$, $c \in L(\texttt{C})$, and $d \in L(\texttt{D})$. By hypothesis, $L(\texttt{BC}{\star}\texttt{D}) \subseteq L(R_1)$, so $q \in L(R_1)$. Similarly, consider another string $r = ef^n g \in L(\texttt{EF}{\star}\texttt{G})$ where $e, g \in \Sigma^*$, $f \in \Sigma^+$, $e \in L(\texttt{E})$, $f \in L(\texttt{F})$, and $g \in L(\texttt{G})$. By hypothesis, $L(\texttt{EF}{\star}\texttt{G}) \subseteq L(R_2)$, so $r \in L(R_2)$. As $L(\texttt{C}) \cap L(\texttt{F}) \cap L(\texttt{DE}) = \phi$, suppose $c = f = de$.

Consider the new string $p = qr = bc^m def^n g \in L(R_1){\cdot}L(R_2) = L(R_1{\cdot}R_2)$. In other words, $R_1{\cdot}R_2$ should include the following NFA accepting p.



For $m = 2$ and $n = 2$, $p = bccdeffg$. There are $(m \times n) + 1 = (2 \times 2) + 1 = 5$ ways to match. Ignoring prefix b and suffix g, the five cases to match the middle $ccdeff$ are:

- $v_1 \rightarrow^{c} v_1 \rightarrow^{c} v_1 \rightarrow^{(de=c)} v_1 \rightarrow^{(f=c)} v_1 \rightarrow^{(f=de)} v_3$
- $v_1 \rightarrow^{c} v_1 \rightarrow^{c} v_1 \rightarrow^{(de=c)} v_1 \rightarrow^{(f=de)} v_3 \rightarrow^{f} v_3$
- $v_1 \rightarrow^{c} v_1 \rightarrow^{c} v_1 \rightarrow^{d} v_2 \rightarrow^{e} v_3 \rightarrow^{f} v_3 \rightarrow^{f} v_3$
- $v_1 \rightarrow^{c} v_1 \rightarrow^{(c=de)} v_3 \rightarrow^{(de=f)} v_3 \rightarrow^{f} v_3 \rightarrow^{f} v_3$
- $v_1 \rightarrow^{(c=de)} v_3 \rightarrow^{c=f} v_3 \rightarrow^{(de=f)} v_3 \rightarrow^{f} v_3 \rightarrow^{f} v_3$

where the superscript of an arrow represents the (input observed = path taken) pair.

The degree of ambiguity grows for each larger m and n. It can be shown that for an input string $p = bc^m def^n g$, there will be $(m \times n) + 1$ ways to match. Here ambiguity is a function of the input length. Therefore, $R_1 \cdot R_2$ is infinitely ambiguous.

□

$\Longrightarrow$: If $R_1 \cdot R_2$ is infinitely ambiguous, then $L(R_1)$ contains the language of a regex `BC*D` and $L(R_2)$ contains the language of a regex `EF*G`, where $\epsilon \notin L(\texttt{C}) \land \epsilon \notin L(\texttt{F}) \land L(\texttt{C}) \cap L(\texttt{F}) \cap L(\texttt{DE}) = \phi$.

Proof. We will reason over an equivalent, ambiguity-preserving, ϵ-free NFA [40]. The NFA of an infinitely ambiguous regex should include either a Polynomial or an Exponential Degree of Ambiguity (PDA, EDA) section [40], as shown in Figure 1.

We first show that if $R_1 \cdot R_2$ is infinitely ambiguous, then the NFA of $R_1 \cdot R_2$ must contain a PDA (Figure 1(a)). R_1 and R_2 are unambiguous, so none of them should have a full EDA. Concatenating two regexes $R_1 \cdot R_2$ cannot create a new self loop of EDA. Thus, $R_1 \cdot R_2$ must contain a PDA.

Consider the two nodes p with the loop π_1 and q with the loop π_3 in Figure 1(a). As R_1 and R_2 are unambiguous, neither R_1 nor R_2 can include both nodes p and q — because then they would be infinitely ambiguous (not unambiguous). Therefore, R_1 and R_2 each should have a part of PDA; and the partition will appear somewhere along the path π_2 as the loops π_1 and π_3 cannot be newly introduced via concatenation.

Each partition of PDA consists of a prefix, a loop, and a suffix, which can be mapped to a regex of the form `PQ*R`. As a PDA is a part of the whole NFA, more generally, we can conclude that (1) $L(R_1)$ contains the language of a regex `BC*D` and (2) $L(R_2)$ contains the language of a regex `EF*G`: *i.e.,* $L(\texttt{BC*D}) \subseteq L(R_1)$ and $L(\texttt{EF*G}) \subseteq L(R_2)$ where $\epsilon \notin L(\texttt{C}) \land \epsilon \notin L(\texttt{F})$.

After concatenation, the full PDA can be represented by a language of the form `BC*DEF*G`, where `C*` is mapped to the first loop π_1, `DE` to the path π_2, and `F*` to the second loop π_3. Let s be the string that meets the PDA path conditions: $label(\pi_1) = label(\pi_2) = label(\pi_3)$. Then, $s \in L(\texttt{C})$ (by $label(\pi_1)$), $s \in L(\texttt{DE})$, and $s \in L(\texttt{F})$. And thus $L(\texttt{C}) \cap L(\texttt{F}) \cap L(\texttt{DE}) = \phi$.

□

Theorem 2(b) follows from elimination with Theorem 0.

B.3.4. Theorem 3: Ambiguity of Star. *Given unambiguous regex R,*

(a) $R*$ *is infinitely ambiguous iff* $\epsilon \in L(R) \lor L(R)\ \Omega\ L(R*) = \phi$.
(b) $R*$ *cannot be finitely ambiguous.*

The components of Theorem 3 follow from Lemma 2.

Lemma 2. *If R* is ambiguous, it is always infinitely ambiguous.*

Proof. We prove this by induction. From the contrapositive of Theorem 0(c), if $R*$ is ambiguous, $L(R) \not\bowtie L(R*) = \phi$. There exists an input string $s = xay$ such that 1) $x, y \in \Sigma^*$, 2) $a \in \Sigma^+$, 3) $x, xa \in L(R)$, 4) $y, ay \in L(R*)$. In other words, there are at least $2 = 2^1$ ways to parse s (*i.e.,* $x \in L(R)$ then $ay \in L(R*)$; or xa then y).

Now consider $ss = (xay)(xay)$. Let $x' = x, a' = a, y' = yxay$ then, $ss = x'a'y'$. Then the following conditions are true: (1) $x', y' \in \Sigma^*$, (2) $a' \in \Sigma^*$, (3) $x', x'a' \in L(R)$, and (4) $y', a'y' \in L(R * RR*) \subset L(R*)$. For each xay there are at least 2 accepting paths. Therefore, for ss there are at least 4=2^2 accepting paths, and the degree of ambiguity grows for each additional concatenation of an s. Therefore, $R*$ is infinitely ambiguous.

□

B.3.5. Theorem 4: Finite to Infinite. *Given a finitely ambiguous regex R, R* is always infinitely ambiguous.*

Proof. If R is finitely ambiguous by definition there exists an input string s for which there will be at least 2 accepting paths. For $R*$, we can increase the length of input string as much as we want because of the $*$. Now for input string ss, there will be at least $4 = 2^2$ accepting paths as we have at least 2 options for each s. By the same logic, for input string $sss....$ where length of the input string is n, there will be at least 2^n accepting paths.

Therefore, $R*$ is infinitely ambiguous. □

B.4. Limitations

Our theorems do not cover the cases when $R1$ and $R2$ are finitely ambiguous. In such scenario, our expectation is that Alternation ($R_1|R_2$) would always yield a finitely ambiguous regex. We also expect that Concatenation ($R_1 \cdot R_2$) would still yield an infinitely ambiguous regex if $L(R_1)$ contains the language of a regex `BC*D` and $L(R_2)$ contains the language of a regex `EF*G`, where $\epsilon \notin L(\texttt{C}) \land \epsilon \notin L(\texttt{F}) \land L(\texttt{C}) \cap L(\texttt{F}) \cap L(\texttt{DE}) = \phi$. However, whether this is the only case is less clear. Still, despite this limitation, our theorems allowed us to derive anti-patterns (§5) and fix strategies (§6) that substantially improved the effectiveness of the SOA ones (§7, §8), and the usability of existing automatic detection and fixing tools (§9).

Appendix C.
Other Figures

C.1. CVEs Increasing Year by Year

We observe annual growth in ReDoS CVEs from 2010 to the present. Figure 4 shows the trend of ReDoS CVEs since 2010. The incidence of ReDoS CVEs grew from 2 in 2010 to 20 in 2021.

C.2. Fix Acceptance

We asked the participants of Experiment 3 (§9) how comfortable they were replacing the vulnerable regex in their codebase with the fixes provided by each repair treatment. As shown in Figure 5, almost all subjects were more comfortable with the fix produced using our anti-patterns and fix strategies.

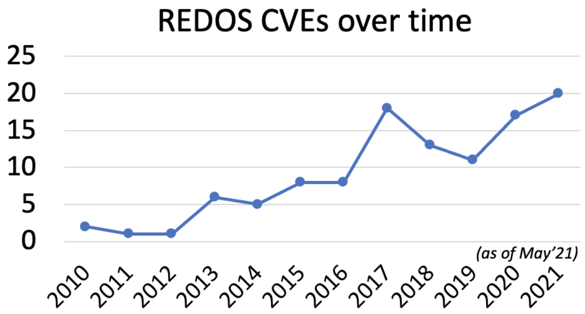


Figure 4: The data were obtained by a two-step process: a preliminary labeling of the CVE database using key words and phrases (*e.g.,* "ReDoS" or "extremely long time" with a reference to regular expressions), followed by a manual inspection for accuracy.

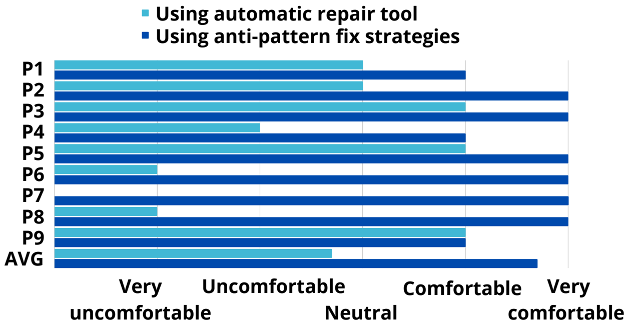


Figure 5: Fixing Task: How comfortable our subjects reported being with fixing their codebase with the fix produced by each treatment.

References

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

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

[3] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish, "Regular expression learning for information extraction," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 08)*, 2008, pp. 21–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=1613715.1613719

[4] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[5] L. Chiticariu, V. Chu, S. Dasgupta, T. W. Goetz, H. Ho, R. Krishnamurthy, A. Lang, Y. Li, B. Liu, S. Raghavan, F. R. Reiss, S. Vaithyanathan, and H. Zhu, "The systemt ide: An integrated development environment for information extraction rules," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 11)*, 2011, pp. 1291–1294. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989479

[6] S. Efftinge and M. Völter, "oaw xtext: A framework for textual dsls," in *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, 2006, p. 118.

[7] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2008, pp. 387–401.

[8] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, 2008, p. 171–180. [Online]. Available: https://doi.org/10.1145/1368088.1368112

[9] "Owasp modsecurity core rule set,," https://coreruleset.org/.

[10] N. L. Or, X. Wang, and D. Pao, "Memory-based hardware architectures to detect clamav virus signatures with restricted regular expression features," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1225–1238, 2016.

[11] S. Crosby, "Denial of service through regular expressions," *USENIX Security work in progress report*, 2003.

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

[13] S. Exchange, "Outage postmortem," http://web.archive.org/web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

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

[15] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *USENIX Security Symposium (USENIX Security)*, 2018. [Online]. Available: https://www.npmjs.com/package/safe-regexhttp://mp.binaervarianz.de/ReDoS_TR_Dec2017.pdf

[16] E. Barlas, X. Du, and J. C. Davis, "Exploiting input sanitization for regex denial of service," in *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, 2022, p. 883–895. [Online]. Available: https://doi.org/10.1145/3510003.3510047

[17] M. Berglund, F. Drewes, and B. Van Der Merwe, "Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching," *EPTCS: Automata and Formal Languages 2014*, vol. 151, pp. 109–123, 2014. [Online]. Available: https://arxiv.org/pdf/1405.5599.pdf

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

[19] N. H. Weideman, "Static Analysis of Regular Expressions," *MS Thesis*, no. December, 2017.

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

[21] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, may 2021, pp. 1468–1484. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00062

[22] J. Kirrage, A. Rathnayake, and H. Thielecke, "Static Analysis for Regular Expression Denial-of-Service Attacks," *Network and System Security*, vol. 7873, pp. 35–148, 2013.

[23] A. Rathnayake and H. Thielecke, "Static Analysis for Regular Expression Exponential Runtime via Substructural Logics," *CoRR*, 2014.

[24] S. Sugiyama and Y. Minamide, "Checking Time Linearity of Regular Expression Matching Based on Backtracking," *Information and Media Technologies*, vol. 9, no. 3, pp. 222–232, 2014.

[25] M. Sulzmann and K. Z. M. Lu, "Derivative-Based Diagnosis of Regular Expression Ambiguity," *International Journal of Foundations of Computer Science*, vol. 28, no. 5, pp. 543–561, 4 2017. [Online]. Available: http://arxiv.org/abs/1604.06644

[26] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, "Rescue: Crafting regular expression dos attacks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 18)*, 2018, p. 225–235. [Online]. Available: https://doi.org/10.1145/3238147.3238159

[27] B. Van Der Merwe, N. Weideman, and M. Berglund, "Turning Evil Regexes Harmless," in *South African Institute of Computer Scientists and Information Technologists (SAICSIT)*, 2017. [Online]. Available: https://doi.org/10.1145/3129416.3129440

[28] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, and M. O'Neill, "A search for improved performance in regular expressions," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 17)*, 2017, p. 1280–1287. [Online]. Available: https://doi.org/10.1145/3071178.3071196

[29] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S.-C. Cheung, and H. Zhao, "Flashregex: deducing anti-redos regexes from examples," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 659–671.

[30] M. Claver, J. Schmerge, J. Garner, J. Vossen, and J. McClurg, "Regis: Regular expression simplification via rewrite-guided synthesis," *arXiv preprint arXiv:2104.12039*, 2021.

[31] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[32] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[33] S. A. Hassan, "Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies," Dec. 2022. [Online]. Available: https://zenodo.org/badge/latestdoi/575922405

[34] S. Kleene, "Representation of events in nerve nets and finite automata," RAND PROJECT AIR FORCE SANTA MONICA CA, Tech. Rep., 1951.

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

[36] J. E. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2002.

[37] J. C. Davis, D. Moyer, A. M. Kazerouni, and D. Lee, "Testing Regex Generalizability And Its Implications: A Large-Scale Many-Language Measurement Study," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[38] C. Brabrand and J. G. Thomsen, "Typed and unambiguous pattern matching on strings using regular expressions," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming (PPDP 10)*, 2010, pp. 243–254.

[39] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM journal of research and development*, vol. 3, no. 2, pp. 114–125, 1959.

[40] A. Weber and H. Seidl, "On the degree of ambiguity of finite automata," *Theoretical Computer Science*, vol. 88, no. 2, pp. 325–349, 1991.

[41] C. Allauzen, M. Mohri, and A. Rastogi, "General algorithms for testing the ambiguity of finite automata," in *International Conference on Developments in Language Theory*. Springer, 2008, pp. 108–120.

[42] R. E. Stearns and H. B. Hunt III, "On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata," *SIAM Journal on Computing*, vol. 14, no. 3, pp. 598–611, 1985.

[43] J. C. Davis, F. Servant, and D. Lee, "Using selective memoization to defeat regular expression denial of service (redos)," in *2021 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA*, 2021, pp. 543–559.

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

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

[46] L. G. Michael IV, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are Hard : Decision-making, Difficulties, and Risks in Programming Regular Expressions," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[47] J. C. Davis, "On the impact and defeat of regular expression denial of service," Ph.D. dissertation, Virginia Tech, 2020.

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

[49] L. Turoňová, L. Holík, I. Homoliak, O. Lengál, M. Veanes, and T. Vojnar, "Counting in regexes considered harmful: Exposing ReDoS vulnerability of nonbacktracking matchers," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4165–4182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/turonova

[50] G. R. Bai, B. Clee, N. Shrestha, C. Chapman, C. Wright, and K. T. Stolee, "Exploring tools and strategies used during regular expression composition tasks," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 197–208.

[51] P. Wang, G. R. Bai, and K. T. Stolee, "Exploring regular expression evolution," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 502–513.

[52] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S.-C. Cheung, "{ReDoSHunter}: A combined static and dynamic approach for regular expression {DoS} detection," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3847–3864.

[53] R. McLaughlin, F. Pagani, N. Spahn, C. Kruegel, and G. Vigna, "Regulator: Dynamic analysis to detect ReDoS," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022, pp. 4219–4235. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/mclaughlin

[54] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *Computer and Communications Security (CCS)*, 2017. [Online]. Available: https://arxiv.org/pdf/1708.08437.pdf

[55] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 18)*, 2018, pp. 213–223.

[56] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: complexity analysis with fuzzing and symbolic execution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 18)*, 2018, pp. 322–332.

[57] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 393–410.

[58] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," *arXiv preprint arXiv:2002.03416*, 2020.

[59] N. Chida and T. Terauchi, "Repairing dos vulnerability of real-world regexes," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2060–2077.

[60] Z. Bai, K. Wang, H. Zhu, Y. Cao, and X. Jin, "Runtime recovery of web applications under zero-day redos attacks," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1575–1588.

[61] N. Atre, H. Sadok, E. Chiang, W. Wang, and J. Sherry, "Surgeprotector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling," in *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), New York, NY, USA*, 2022.

[62] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for javascript and node. js: First-class timeouts as a cure for event handler poisoning," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 343–359.

[63] H. M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B. T. Loo, and L. T. X. Phan, "Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame," in *2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019, pp. 693–708.

[64] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.

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

[66] R. Cox, "Regular Expression Matching in the Wild," 2010. [Online]. Available: https://swtch.com/~rsc/regexp/regexp3.html

[67] T. R. P. Developers, "regex - rust," https://docs.rs/regex/1.1.0/regex/.

[68] Google, "regexp - go," https://golang.org/pkg/regexp/.

[69] O. Saarikivi, M. Veanes, T. Wan, and E. Xu, "Symbolic regex matcher," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 372–378.

[70] L. Holík, O. Lengál, O. Saarikivi, L. Turoňová, M. Veanes, and T. Vojnar, "Succinct determinisation of counting automata via sphere construction," in *Asian Symposium on Programming Languages and Systems*. Springer, 2019, pp. 468–489.

[71] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar, "Regex matching with counting-set automata," in *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 20)*, Virtual, November 2020. [Online]. Available: https://doi.org/10.1145/3428286

[72] S. Sung, H. Cheon, and Y.-S. Han, "How to settle the redos problem: Back to the classical automata theory," in *Implementation and Application of Automata*, P. Caron and L. Mignot, Eds. Cham: Springer International Publishing, 2022, pp. 34–49.

[73] A. Møller, "dk. brics. automaton–finite-state automata and regular expressions for java," 2010.

[74] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, 2020.

[75] substack and Davis, "safe-regex," https://www.npmjs.com/package/safe-regex, 2013.

[76] "antlr-pcre," https://web.archive.org/web/20210826063830/https://github.com/bkiers/pcre-parser.

[77] K. M. Ting, *Precision and Recall*. Boston, MA: Springer US, 2010, pp. 781–781. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_652

[78] N. Chida and T. Terauchi, "Automatic repair of vulnerable regular expressions," *arXiv preprint arXiv:2010.12450*, 2020.

[79] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[80] P. D. Ellis, *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge university press, 2010.

[81] "Pypi – the python package index," https://pypi.python.org/pypi.

[82] "npm," https://www.npmjs.com.

[83] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 63–92.

[84] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299–1334, 2014.

[85] A. Cha, E. Wittern, G. Baudart, J. C. Davis, L. Mandel, and J. A. Laredo, "A principled approach to graphql query cost analysis," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 20)*, 2020, pp. 257–268.