SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS)

Masudul Hasan Masud Bhuiyan[*]
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
masudul.bhuiyan@cispa.de

Berk Çakar[*]
Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
bcakar@purdue.edu

Ethan H. Burmane
Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
eburmane@purdue.edu

James C. Davis
Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
davisjam@purdue.edu

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
staicu@cispa.de

Abstract

Regular Expression Denial of Service (ReDoS) is a vulnerability class that has become prominent in recent years. Attackers can weaponize such weaknesses as part of asymmetric cyberattacks that exploit the slow worst-case matching time of regular expression (regex) engines. In the past, problematic regexes have led to outages at Cloudflare and Stack Overflow, showing the severity of the problem. While ReDoS has drawn significant research attention, there has been no systematization of knowledge to delineate the state of the art and identify opportunities for further research. In this paper, we describe the existing knowledge on ReDoS. We first provide a systematic literature review, discussing approaches for detecting, preventing, and mitigating ReDoS vulnerabilities. Then, our engineering review surveys the latest regex engines to examine whether and how ReDoS defenses have been realized. Combining our findings, we observe that (1) in the literature, almost no studies evaluate whether and how ReDoS vulnerabilities can be weaponized against real systems, making it difficult to assess their real-world impact; and (2) from an engineering view, many mainstream regex engines have introduced partial or full ReDoS defenses, rendering many threat models obsolete. We conclude by highlighting avenues for future work. The open challenges in ReDoS research are to evaluate emerging defenses and support engineers in migrating to defended engines. We also highlight the parallel between performance bugs and asymmetric DoS, and we argue that future work should capitalize more on this similarity and adopt a more systematic view on ReDoS-like vulnerabilities.

CCS Concepts

• **General and reference** → *Surveys and overviews*; • **Security and privacy** → *Denial-of-service attacks*; • **Software and its engineering** → *Empirical software validation*.

[*]Both authors contributed equally to this research.

Keywords

Systematization of knowledge (SoK), regular expression denial of service (ReDoS), regex engines, ReDoS defenses

ACM Reference Format:
Masudul Hasan Masud Bhuiyan, Berk Çakar, Ethan H. Burmane, James C. Davis, and Cristian-Alexandru Staicu. 2025. SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS). In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3708821.3733912

1 Introduction

Regular Expression Denial of Service (ReDoS) [28, 56, 108] is a type of denial of service (DoS) attack [53, 93] that exploits inefficiencies in regular expression (regex) engines. Many regex engines have worst-case exponential backtracking behavior. By crafting input strings that trigger this behavior, attackers can cause systems to consume disproportionate CPU resources, leading to service disruptions. ReDoS threatens real systems: slow regex matches caused outages at Stack Overflow [121] and Cloudflare [57]. ReDoS is the fourth most common server-side vulnerability class in JavaScript's npm ecosystem—only path traversal, prototype pollution, and command injection appear more often [9]. It is also among the fastest-growing vulnerabilities in the same ecosystem [76]. Consequently, there has been much research on ReDoS, with dozens of papers since 2015 (§3). However, as yet there has been no systematization of knowledge about ReDoS, making it challenging to consolidate existing knowledge, identify gaps, and guide future efforts to address this complex and evolving threat.

To address this gap, we conducted a comprehensive literature review of recent papers on ReDoS (§3). Our goals were to provide a tutorial on regexes and ReDoS, summarize previous work, assess progress in the field, and identify future work opportunities. We analyze 35 papers on ReDoS vulnerabilities from top security conferences, focusing on detection, prevention, and mitigation strategies, to provide a comprehensive overview of the current state of the field. One primary observation from our review was that almost all prior works rely on at least one of two main models of regex engine behavior: the first introduced in 1968 by Thompson [130], and the second in 1994 by Spencer [119]. While these models have been instrumental in shaping foundational research, they do not fully reflect modern regex engines, many of which now incorporate defenses against ReDoS attacks. ReDoS research must be situated within real regex engine implementations, motivating us to perform an updated engineering review.

Thus, we conducted an engineering review of modern regex engines to address this disconnect between academic research and practical applications and update the understanding of the capabilities of regex engines (§4). This review examines the implementations in the major programming languages to evaluate how they handle ReDoS vulnerabilities, either through new algorithms, resource caps, or other mitigations. By analyzing these engines' designs, defenses, and real-world adoption, we provide a detailed account of the current state of regex processing and its implications for ReDoS research. This engineering perspective complements the findings of our literature review and highlights the differences between traditional regex engines and contemporary regex engine behavior, offering up-to-date insights for future studies.

As a result of our synthesis of academic knowledge and engineering analysis, we discuss five main findings (§5). In our review of the academic literature, we identified areas where threat models, ReDoS definitions, and attack evaluations could benefit from further refinement and alignment. In our engineering review of regex engines, we note that four major engines have used three distinct algorithms to mitigate ReDoS, but these defenses do not always reduce the time complexity to linear. We encourage the research community to explore opportunities to improve conceptual clarity in future ReDoS research and to focus efforts on evaluating and advancing state-of-the-art ReDoS defenses.

Our primary contributions are:

- We analyze 35 papers on ReDoS vulnerabilities from top security conferences, critiquing their definitions, methodologies, and assumptions. By identifying gaps in the literature, we provide a foundation for more practical and impactful research.
- We examine the current posture of the major regex engines, highlighting recent improvements, persistent vulnerabilities, and the trade-offs involved in mitigating ReDoS.
- Based on our findings, we propose guidelines for researchers to align their work with real-world needs and for practitioners to better evaluate and mitigate ReDoS risks.

2 Background and Motivation

In this section, we describe regexes, regex engines, and ReDoS.

2.1 Regexes

2.1.1 Kleene's regexes (K-regexes). Regexes are a popular technology for string matching tasks [51]. Regexes are used in two basic operations: recognition ("Does this regex match this string?") and parsing ("Extract relevant substrings from this match") [66].

Regexes began with Kleene's [69] "regular" notation (*K-regexes*), which specified a language as a set of strings using core constructs of concatenation (.), disjunction (|), and unbounded repetition (*). These operations define whether a given string belongs to the set of strings described by the regex. Formally, a K-regex R is:

$$R \longrightarrow \emptyset \,\Big|\, \epsilon \,\Big|\, \sigma \,\Big|\, R* \,\Big|\, R_1 \mid R_2 \,\Big|\, R_1 \cdot R_2$$

where the empty language is $\emptyset$, the empty string is ϵ, and the alphabet is Σ. A regex's semantics are defined by its *language*, the set of strings it describes. The language function $L : R \longrightarrow 2^{\Sigma^*}$ is:

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

K-regexes can be modeled with nondeterministic or deterministic finite automata (NFA and DFA) [85, 130]. These automata are 5-tuples $\langle Q, q_0, F, \Sigma, \delta \rangle$, where Q is the set of states, q_0 is the start state, F is the set of accepting states, Σ is the input alphabet, and δ is the transition function. Each regex operator—character matching, concatenation, repetition, and disjunction—has a corresponding NFA representation, as illustrated in Figure 1. These NFAs can be further converted into DFAs, providing deterministic evaluation at the cost of potentially higher state complexity [105].

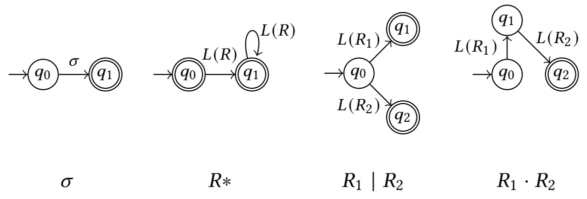


Figure 1: K-regex operators and NFA equivalents, following the Thompson-McNaughton-Yamada construction [85, 130].

2.1.2 Extended regexes (E-regexes). Over time, to simplify the representation, the K-regexes were "extended" (*E-regexes*) with additional operators and constructs. *Syntax sugar* notations, like character ranges (*e.g.*, [A-Z] for A|B|...|Z) and one-or-more-repetition (R+), are still regular and can be transformed into equivalent K-regexes. Some more complex extensions, such as atomic groups [8] and lookarounds (*e.g.*, [7, 14, 52]), have also been shown to be regular.

Many other extensions have been made [51, 64], including irregular features such as backreferences [1]. These extensions increase expressive power but exceed the boundaries of regular languages (*i.e.*, they cannot be directly modeled with finite automata), complicating theoretical analysis, implementation, and matching performance. A curated set of regex features is given in Appendix A.

2.2 Regex Engines

Regexes are supported in all mainstream programming languages [51] and are part of many other systems (*e.g.*, intrusion detection and prevention systems [107, 126], web application firewalls [3], and logging tools [12, 13]). These systems all implement or integrate a component called a *regex engine* that provides support for the regex tasks of recognition and parsing. As summarized by Davis [33], typical regex engine implementations followed one of two algorithms. The first is a depth-first, backtracking search based on a library by Spencer [119]. The second is a breadth-first, lockstep search proposed by Thompson [130].

The characteristics of these engine styles are summarized in Table 1. For the simple case of K-regexes, both can be modeled

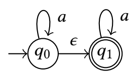
Table 1: Tradeoffs of Spencer and Thompson algorithms for regex engines [27, 33]. Thompson's algorithm uses an efficient algorithm over an automata representation to obtain linear time complexity (in input string length), but this formal approach reduces its expressiveness. Spencer's algorithm gains broader E-regex support at the cost of exponential match times.

Algorithm	Strategy	Match time	Expressiveness
Thompson's	BFS NFA sim.	Linear match	Limited E-regex support
Spencer's	Backtracking	Exponential match	Easy E-regex support

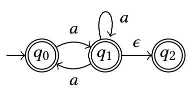
as NFA simulations.[1] These approaches differ in their handling of *ambiguity* [2]—where multiple transitions in the NFA are possible.

- Thompson's algorithm resolves ambiguity by tracking the set of all states in which the automata might be, updating this set using the transition function δ on each new character.
- Spencer's algorithm resolves ambiguity by trying one path, and backtracking to try the other(s) if the first path fails.

Most of the time (for most regexes, inputs, and application contexts), both approaches run quickly enough and do not form a bottleneck for the broader application. However, while the Thompson's algorithm approach guarantees match times that are linear in the input length, Spencer's algorithm exhibits worst-case polynomial or exponential match times on certain regex-input combinations. Figure 2 gives two examples where Spencer's algorithm exhibits polynomial and exponential time complexity, respectively. In each case, on an input $w = aa...ab$, the regex engine encounters ambiguity when processing an "a" symbol. At the ultimate mismatch (caused by the final "b"), the backtracking will take $|w|^2$ time for example (a) and $2^{|w|}$ time for example (b). This worst-case behavior is colloquially termed *catastrophic backtracking*. Note that the actual cost of the match depends on the length of the input; the term *pumping* refers to the repetition of the problematic substring of the input (each "a" in this example).



(a) NFA for $a{*}a{*}$



(b) NFA for $(aa{*}){*}$

Figure 2: Example regexes that, in Spencer's algorithm, exhibit time complexity that is either: (a) polynomial or (b) exponential; in the length of the input string $w = aa...ab$.

Historically, regex engine maintainers accepted this worst-case behavior as the cost of supporting all E-regex features. These extended features are relatively easy to implement in a (Spencer-style) backtracking engine, which often takes the form of a recursive descent parser [38]. New E-regex features can be added by extending the set of parse operators, or equivalently by extending the meaning of the automaton's edge definitions [33]. For example, zero-width assertions like `\b` can use specialized ϵ-edges that depend on passing a contextual test. Backreferences can introduce edges that validate group content while advancing the input position accordingly. Similarly, lookaround assertions can apply recursion to evaluate sub-patterns within the backtracking framework [38]. These implementations are possible because Spencer's algorithm processes paths sequentially, which simplifies keeping track of path-specific behavior.

[1]Implementations of Thompson's algorithm typically have a direct mapping to an NFA simulation, while implementations of Spencer's algorithm may be implemented as recursive descent parsers. Here, we treat K-regexes, where both amount to NFA simulations.

2.3 Regular Expression Denial of Service (ReDoS)

2.3.1 ReDoS in Principle. Regex engines often favor expressive power over predictable worst-case efficiency, and that engineering trade-off leaves them open to ReDoS [28]. The Common Weakness Enumeration catalog labels the problem CWE-1333: *Inefficient Regular Expression Complexity* [88]. ReDoS is an example of an algorithmic complexity attack [29], characterized by the small cost to create a malicious input and the large cost to process it. Algorithmic complexity attacks are one kind of asymmetric DoS attacks [83], where the asymmetry comes from the worst-case performance of the data structures and algorithms used to implement the system.

Following Davis *et al.*'s [38] definition, the threat model for a ReDoS attack has three ingredients:

1. *A Backtracking Regex Engine:* The regex engine employed for pattern matching in the victim system utilizes a backtracking approach (§2.2). Many regex engines implemented in programming languages (*e.g.*, JavaScript, Java, Python) have at some point met this criterion (§4).
2. *A Vulnerable Regex:* The victim system uses a *super-linear, problematically ambiguous regex*; for which the attacker crafts *attack strings* that take polynomial or exponential time to match in the string length [139, 141].
3. *An Adversary-Controlled Input:* The attacker supplies inputs that result in attack strings reaching the vulnerable regex and triggering its worst-case time complexity.

Given these three ingredients, if the system has insufficient mitigations (safeguards) in place, then the attacker can cause the victim system to consume excessive computational resources (*e.g.*, CPU and memory). The attacker may need to tailor the number of pumps based on the length of typical input (to avoid anomaly detection) or the maximum length allowed (to avoid truncation).

2.3.2 ReDoS in Practice. In recent years, ReDoS has become a common security concern. Two major web services had outages that resulted from slow regex behavior: Stack Overflow in 2016 [121] and Cloudflare in 2019 [57]. While these outages did not involve an adversary (*i.e.*, they were reliability issues, not security issues), they made service providers aware of the risk of ReDoS. Since 2020, the security company Snyk has observed a jump in reported ReDoS vulnerabilities, especially in the JavaScript package ecosystem npm [76]. In parallel with this surge, a rich body of grey literature has emerged to offer practical guidance on repairing ReDoS vulnerabilities [4, 30, 55, 78].

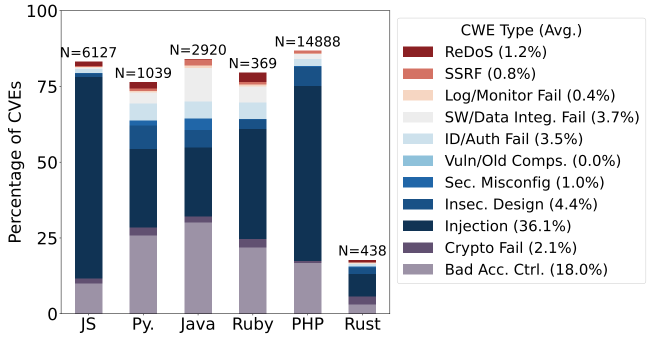


Figure 3: Distribution of OWASP Top 10 and ReDoS CVEs per ecosystem, showing average prevalence per category. N indicates the total number of CVEs recorded for each ecosystem. The data cover 2014-2023. For details, see Appendix B.

For a sense of the formal disclosure of ReDoS vulnerabilities, we examined the National Vulnerability Database (NVD) for Common Vulnerabilities and Exposures (CVE) records pertaining to ReDoS. Following the keyword method of Hassan *et al.* [63], since 2014 ~400 ReDoS CVEs have been disclosed in software from six major programming language ecosystems. As a reference point, we compared these results to CVEs related to the OWASP Top 10 [43]. Averaging over 2014-2023, Figure 3 shows that the rate of ReDoS exceeds four of the OWASP Top 10 vulnerability types.

3 Academic Research on ReDoS

Contemporaneous with the increasing industry recognition of ReDoS, ReDoS has become a common subject of cybersecurity research. However, there has been no systematization of the resulting knowledge. We therefore conducted a systematic literature review of ReDoS papers published between 2015 and 2024 at leading conferences in computer security (S&P, USENIX, CCS, NDSS), software engineering (ICSE, FSE, ASE), and programming languages (PLDI, POPL, OOPSLA). Following the structured methodology outlined by Schloegel *et al.* [111], we identified relevant papers in conference proceedings via an initial keyword search conducted in June 2024, using the terms `ReDoS`, `denial-of-service`, `regex`, `regular expression`, `regular expression analysis`, and `regular expression execution`. This yielded 93 papers, which were further filtered via the following criteria:

- **Inclusion Criteria:** Papers that primarily address ReDoS, propose new attacks or defenses against ReDoS, or extensively study existing ReDoS attacks or defenses.
- **Exclusion Criteria:** Papers that focus on general DoS or those that discuss regexes, without specifically addressing ReDoS. Appendix C provides further discussion of the excluded works.

To ensure informed inclusion and exclusion decisions, we reviewed the abstract, introduction, and evaluation sections of each paper. We identified 35 papers that met our inclusion criteria and systematically analyzed them along four key dimensions:

1. **Research Directions**: Captures the primary focus of each study, which typically falls into one of three categories:
    - *Detection*: Identifying and analyzing vulnerabilities.
    - *Prevention*: Addressing root causes by improving regex engines or limiting unsafe constructs.
    - *Mitigation*: Reducing attack impact through runtime defenses or repair mechanisms.
2. **Threat Models**: Examines the assumptions about attacker capabilities (*e.g.*, control over input) and system behavior (*e.g.*, rate-limiting or regex engine design), highlighting the practical applicability of each study.
3. **Vulnerability Definitions**: Focuses on how papers describe what constitutes a ReDoS vulnerability or a vulnerable regex (*e.g.*, having exponential matching time).
4. **Evaluation Methods**: Assesses the languages and regex engines used for testing, providing insights into the platforms where these approaches are validated.

Additionally, we reviewed other closely related works that, while not meeting our strict inclusion criteria, are still highly relevant to ReDoS. For instance, papers such as [27, 65, 117] propose improved regex matching algorithms. Although mitigating ReDoS was not their primary focus, these works provide valuable insights and serve as foundational building blocks for ReDoS defenses, making them significant for academics and researchers in the field.

We begin by discussing the main existing research directions (§3.1), distinguishing between detection, prevention, mitigation, and other related works. We then explore the assumptions made about the attackers' capabilities or the underlying systems (§3.2), along with the definitions of ReDoS vulnerabilities or successful attacks provided in prior works (§3.3). Finally, we discuss how the studied papers were evaluated (§3.4).

3.1 Main Existing Research Directions

Figure 4 provides an overview of the main research directions in the study of ReDoS vulnerabilities, categorized into detection, prevention, mitigation, and complementary studies. Detection focuses on identifying ReDoS vulnerabilities in regexes and understanding their exploitability. Prevention aims to address the root causes of ReDoS vulnerabilities by improving regex engines or restricting vulnerable constructs. Mitigation, on the other hand, focuses on minimizing the impact of ReDoS attacks by deploying runtime safeguards. We discuss these categories in detail below.

3.1.1 Detection. Detection is a critical area of research in ReDoS, with studies focusing on identifying vulnerable regexes across various ecosystems. Nine papers analyze open-source package ecosystems, such as npm [9, 120], pip [36, 138], and Maven [73, 138], while seven focus on testing live web applications [6, 120]. Additionally, 27 papers investigate ReDoS vulnerabilities using curated regex datasets [79, 115, 116, 134]. We categorize detection methods into five key approaches: *Heuristic-based Detection* (three papers), *Automata-based Detection* (seven papers), *Dynamic Detection* (four papers), *Hybrid Detection* (five papers), and *Machine Learning (ML)-based Detection* (one paper). Heuristic-based techniques rely on simple pattern rules to flag problematic regexes, offering speed but often at the cost of high false positive rates. Automata-based approaches, which model regexes using finite-state machines, provide formal accuracy but are computationally expensive. Dynamic detection evaluates regex behavior on specific inputs, revealing real-world vulnerabilities but requiring significant runtime resources.

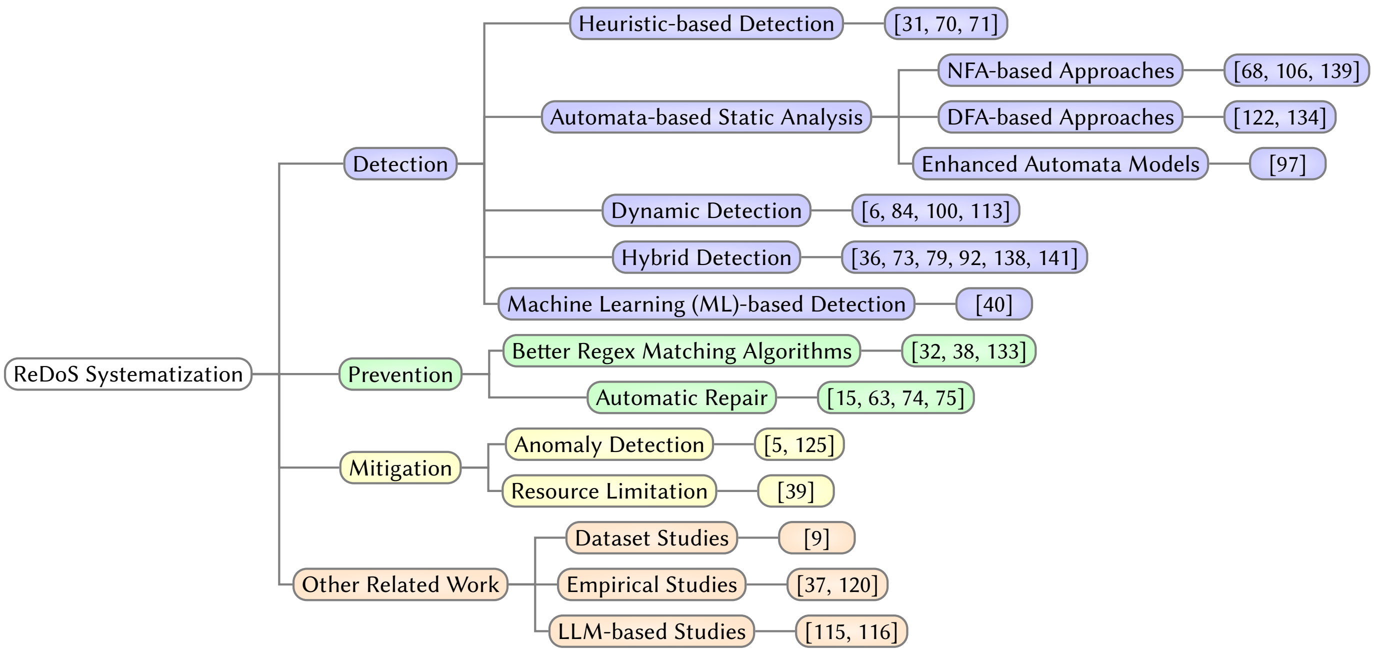


Figure 4: Systematization of ReDoS research papers categorized into detection, prevention, mitigation, and related studies, with subcategories and key references.

Hybrid methods combine static and dynamic techniques to balance precision and efficiency, while emerging ML approaches use regex datasets to predict vulnerabilities, opening new avenues but facing challenges like adversarial examples and limited training data. This categorization highlights the trade-offs in speed, accuracy, and false positive/negative rates among different techniques, offering a structured view of existing detection strategies and their potential for future enhancement.

Heuristic-based Detection (3 studies): There are many approaches that employ simple pattern-matching techniques to identify potentially vulnerable regexes. Studies by Kluban *et al.* [70, 71] and Davis *et al.* [31] utilize tools like regexploit [80], redos-detector [67], and safe-regex [34] to identify these vulnerabilities by searching for constructs such as infinite repeats (`a*b*a*`), branches ($a|b$), or nested quantifiers (`*`, `+`, `?`, `{n,m}`), which are often associated with ReDoS vulnerabilities. While these methods are fast, enabling the rapid analysis of large numbers of regexes, they typically focus on syntax rather than semantics, leading to potential accuracy issues. For instance, Parolini *et al.* [97] demonstrated that while tools like regexploit achieved low false positive and negative rates, others like safe-regex and redos-detector exhibited significantly higher false positive rates. This highlights the primary limitation of heuristic-based approaches, which often require further analysis to confirm vulnerabilities.

Automata-based Static Analysis (6 studies): Automata-based static analysis is a powerful approach for detecting problematic regexes without requiring program execution. This method has been explored in several studies [68, 97, 106, 122, 134, 139], which employ automata models such as NFAs, DFAs, and enhanced automata to evaluate regex semantics and detect vulnerabilities that lead to ReDoS attacks.

NFA-based methods, such as those by Kirrage *et al.* [68] and Rathnayake *et al.* [106], construct NFAs to analyze patterns like (`prefix, pumpable string, suffix`) and identify vulnerabilities caused by exponential backtracking. Building on this, Weideman *et al.* [139] enhance NFAs with prioritization (pNFA) to reduce redundant state exploration, providing a more precise analysis of backtracking vulnerabilities.

DFA-based approaches, as demonstrated by Turoňová *et al.* [134] and Shen *et al.* [122], rely on deterministic automata simulations to evaluate vulnerabilities in non-backtracking engines. By incorporating specialized counting mechanisms, these approaches effectively detect problematic patterns, such as bounded quantifiers and repetitive constructs, while optimizing resource usage. This focus on determinism allows for efficient analysis of regexes without the ambiguity associated with backtracking.

Enhanced automata models offer a novel perspective on regex mathing. Parolini *et al.* [97] introduce tree semantics, extending beyond traditional NFA and DFA approaches by capturing the structural interactions of regex engines. This enables the detection of vulnerabilities that are difficult to represent in standard automata.

While these methods excel in early detection, they face challenges such as false positives from over-approximations, false negatives from under-approximations, and high computational complexity for analyzing intricate regex structures. Despite these trade-offs, automata-based static analysis remains a foundational tool for addressing ReDoS vulnerabilities.

Dynamic Analysis (4 studies): Dynamic approaches [6, 84, 100, 113] to detecting ReDoS vulnerabilities involve executing regexes with various inputs and analyzing their runtime behavior to identify potential issues. For instance, Shen *et al.* [113] developed ReScue, a tool that uses an evolutionary genetic algorithm to generate time-consuming strings capable of triggering ReDoS vulnerabilities. Similarly, Barlas *et al.* [6] and McLaughlin *et al.* [84] utilized dynamic analysis with fuzzing techniques, while Petsios *et al.* [100] introduced SlowFuzz, which uses automated fuzzing to find inputs causing worst-case algorithmic behavior.

Hybrid Approaches (6 studies): Existing static analysis methods for detecting ReDoS vulnerabilities often face a trade-off between precision and recall. For example, Davis *et al.* [36] and Shen *et al.* [113] reported low F-1 scores of 44.94% and 3.57%, respectively [73]. To address these limitations, dynamic validation is frequently employed alongside static analysis, resulting in hybrid approaches. Tools such as NFAA [141], Revealer [79], ReDoSHunter [73], Badger [92], and RENGAR [138] combine static and dynamic methods to mitigate false positives. For instance, hybrid approaches like ReDoSHunter integrate static and dynamic techniques to improve detection accuracy, while tools such as NFAA refine dynamic testing with insights from static analysis [141]. These methods effectively balance the strengths of both static and dynamic detection, reducing the likelihood of inaccurate results.

Machine Learning (ML)-based Detection (1 study): ML techniques are also gaining traction in ReDoS detection. Unlike hybrid methods, which explicitly integrate static and dynamic analysis, ML approaches aim to predict vulnerabilities and detect anomalies based on patterns in data. For example, Demoulin *et al.* [40] introduced a semi-supervised learning model to analyze resource utilization and identify potential vulnerabilities. This method triggers alerts and provides mitigation strategies by detecting anomalies in resource consumption. While ML approaches offer promise, they face challenges such as reliance on labeled data, vulnerability to adversarial inputs, and difficulty in addressing context-specific patterns that may not generalize well across diverse regex use cases.

3.1.2 *Prevention.*

Prevention strategies focus on addressing the root causes of ReDoS vulnerabilities, either by improving regex matching algorithms or by repairing vulnerable regex patterns. These approaches aim to prevent ReDoS attacks entirely by designing systems that eliminate worst-case performance scenarios, thus avoiding the need for runtime detection or mitigation.

Better Regex Matching Algorithms (3 studies): Researchers have proposed developing new regex matching engines to address ReDoS vulnerabilities at a fundamental level. Davis *et al.* [32, 38] introduced memoization-based optimization to speed up regex matching by caching intermediate results, significantly reducing redundant computations. Turoňová *et al.* [133, 134] proposed novel counting automata matching algorithms that further optimize counter management during matching to efficiently handle regexes with bounded repetitions. These new engines often employ finite-state machine-based approaches instead of backtracking, inherently avoiding the exponential time complexity that leads to ReDoS vulnerabilities. Additionally, some methods impose restrictions on regex features to ensure faster and safer processing.

Automatic Repair (4 studies): Automatic repair techniques aim to proactively address vulnerabilities in regexes to mitigate ReDoS attacks. Several studies have explored programmatic frameworks to detect and repair problematic regex patterns. For example, Li *et al.* [75] developed FlashRegex, a programming-by-example (PbE) tool that repairs or synthesizes regex from provided matching examples. However, FlashRegex struggles with complex regex features, such as lookarounds and backreferences [74]. To address these limitations, RegexScalpel [74] employs a localize-and-fix approach, which identifies fine-grained vulnerabilities and applies predefined repair strategies to improve recall and precision. In a complementary effort, Hassan *et al.* [63] focus on identifying specific regex structures or "anti-patterns" that are prone to ReDoS vulnerabilities. Their method analyzes regex patterns to uncover and repair potential vulnerabilities early in the development process, avoiding the need for runtime execution. This approach complements programmatic frameworks by systematically identifying and addressing problematic constructs.

Other tools, such as Chida *et al.* [15], also leverage PbE algorithms but face challenges related to dependency on user-provided examples, which can result in incomplete repairs [73, 138]. These methods, while effective for certain regex patterns, often struggle with advanced or highly customized regex constructs, underscoring the need for more comprehensive solutions capable of addressing complex vulnerabilities. Moreover, improving the generalization and robustness of automatic repair frameworks remains a key area of ongoing research.

Prevention techniques address ReDoS vulnerabilities by improving the efficiency and resilience of regex engines or by repairing vulnerable regex patterns. While they significantly reduce the risk of attacks, these methods require careful configuration to ensure compatibility with complex patterns and real-world scenarios.

3.1.3 *Mitigation.*

Mitigation strategies aim to detect ReDoS attacks in live systems and mitigate their impact in real-time. These approaches focus on monitoring runtime behavior to identify anomalies in resource consumption, such as excessive memory usage or prolonged processing times, and applying defensive measures to minimize the effects of ongoing attacks. Unlike prevention, which aims to eliminate vulnerabilities before execution, mitigation provides a dynamic response to attacks as they occur.

Anomaly Detection (2 studies): Anomaly detection techniques are central to mitigation strategies, leveraging real-time monitoring to identify unusual patterns indicative of ReDoS attacks. ML-based approaches have proven effective in this domain [5, 125]. For example, Tandon *et al.* [125] employ the Elliptic Envelope model to analyze runtime features such as request time, memory usage, and CPU cycles, enabling the system to detect deviations from normal behavior. Bai *et al.* [5] introduce a feedback loop mechanism that continuously adjusts defensive measures based on real-time performance metrics, providing robust protection against zero-day ReDoS attacks.

While promising, ML-based anomaly detection methods are not without challenges. For instance, Tandon *et al.* [125] report a false positive rate of 1.3% and a false negative rate of 0.4%, which could still impact practical deployment. Furthermore, these systems can be vulnerable to adversarial attacks, where carefully crafted inputs are designed to bypass detection mechanisms [5]. These limitations underscore the need for more resilient and adaptive approaches to anomaly detection, ensuring robustness against both known and emerging threats.

Limiting Resource Consumption (1 study): To mitigate ReDoS attacks, Davis *et al.* [39] proposed using first-class timeouts as a defense against Event Handler Poisoning (EHP) in Node.js. Their prototype, `Node.cure`, effectively limits processing time with minimal performance overhead. Similarly, runtime environments like PHP, Perl, and .NET have adopted resource-limiting mechanisms to curb backtracking and excessive processing. In PHP, the pcre.backtrack_limit and pcre.recursion_limit directives control the number of backtracks and recursion depth during regex matching [101]. Perl's PCRE2 library allows setting evaluation limits via the eval function, with a default recursion depth limit of 10M. In .NET, the RegexOptions.MatchTimeout property enables developers to specify a maximum regex processing duration. While these strategies effectively cap resources to reduce ReDoS risks, they are often too coarse-grained, making it challenging to balance performance and avoid false positives in complex regex patterns or large datasets.

3.1.4 *Other Related Work.*

Beyond approaches for addressing ReDoS directly, several complementary studies provide valuable insights into datasets, real-world analyses, and novel frameworks.

Dataset Studies (1 study): Bhuiyan *et al.* [9] introduced SecBenchJS, a benchmark dataset for evaluating the security of JavaScript applications, including regex vulnerabilities. Such datasets enable standardized testing and improve ReDoS detection tools.

Empirical Studies (2 studies): Staicu *et al.* [120] analyzed 2846 popular websites, finding 339 vulnerable to ReDoS attacks, highlighting the risks of server-side JavaScript. Davis *et al.* [37] studied regex portability across eight programming languages, uncovering semantic and performance differences in 25% of regexes. These studies underscore the need for safer regex practices and consistent regex engine implementations.

LLM-based Studies (2 studies): Siddiq *et al.* [115] proposed Re(gEx|DoS)Eval, a framework for evaluating LLM-generated regexes using metrics like pass@k and vulnerable@k, demonstrating its use on T5, Phi-1.5, and GPT-3.5-Turbo. In another study, Siddiq *et al.* [116] categorized vulnerable regex patterns and analyzed developer discussions, finding that GPT-3.5-Turbo often generates regexes with polynomial vulnerabilities.

These studies provide benchmarks, empirical data, and frameworks that complement prevention, detection, and mitigation strategies, paving the way for robust ReDoS solutions.

3.2 Threat Models

A realistic threat model is essential in any security study, as it provides a framework for assessing the potential findings. That is, it enables the identification of the system's weaknesses (*i.e.*, vulnerabilities) that adversaries could exploit in real-world environments. A strong threat model might lead to unrealistic findings, assuming too powerful attackers. In Table 2, we outline the granularity at which different threat models are defined in the analyzed papers. While most studies examine the exploitability of regexes in isolation, some do extend their analysis to the surrounding context, such as within library code. However, only a few studies define threat models at the application level. Threat models defined at the code fragment level can lead to false positives—vulnerabilities that are exploitable in isolated regexes but unreachable in actual code.

Among the papers we reviewed, only 10 (28%) explicitly define a threat model, while the rest implicitly assume one. We identify four assumptions in the existing threat models:

Table 2: Overview of ReDoS threat models in prior work. Each column shows whether the study analyzes code fragments (standalone regex snippets), use within libraries, or full applications, and whether it assumes the attacker can send an arbitrary number of requests. The symbol ● denotes the paper covers that aspect in detail; – means it does not.

Paper	Code Fragment	Library	Application	Arbitrary Number of Requests
Su *et al.* [122]	●	–	–	100kB
Davis *et al.* [31]	●	–	–	–
Turoňová *et al.* [134]	●	–	–	1.5K–9K
Liu *et al.* [79]	●	–	–	128
Staicu *et al.* [120]	–	●	●	82K
Davis *et al.* [36]	●	●	–	100K–1M
Siddiq *et al.* [116]	●	–	–	–
Shen *et al.* [113]	●	–	–	128
Barlas *et al.* [6]	–	–	●	–
Siddiq *et al.* [115]	●	–	–	–
Davis *et al.* [37]	●	–	–	–
Kluban *et al.* [70]	●	–	–	–
Bhuiyan *et al.* [9]	●	●	–	–
Noller *et al.* [92]	●	–	–	–
Wang *et al.* [138]	●	●	–	–
Rathnayake *et al.* [106]	●	–	–	–
McLaughlin *et al.* [84]	●	●	–	1M
Li *et al.* [73]	●	●	–	–
Parolini *et al.* [97]	●	–	–	128
Kirrage *et al.* [68]	●	–	–	–
Wüstholz *et al.* [141]	●	–	●	140K
Petsios *et al.* [100]	●	–	–	–
Kluban *et al.* [71]	–	●	–	–
Demoulin *et al.* [40]	–	–	●	–
Chida *et al.* [15]	●	–	–	–
Bai *et al.* [5]	–	–	●	50K
Li *et al.* [74]	●	●	–	1M
Davis *et al.* [32]	●	–	–	–
Li *et al.* [75]	●	–	–	–
Hassan *et al.* [63]	●	–	–	–
Davis *et al.* [39]	–	●	●	–
Davis *et al.* [38]	●	–	–	10K–100K
Tandon *et al.* [125]	–	–	●	–
Weideman *et al.* [139]	●	–	–	–
Turoňová *et al.* [133]	●	–	–	500K–50M

Attacker Controls Input (35 studies): The assumption that an attacker can control inputs to a vulnerable ReDoS location is crucial for exploitability, as highlighted in various studies [6, 63]. In real-world software, including web applications and systems, inputs often originate from untrusted sources, thereby increasing the potential for exploitation [100]. However, whether these inputs can reach the vulnerable regex depends on the surrounding code. All the reviewed papers, whether their threat models focus on the application [6, 40, 120, 141], library [9, 73, 84, 138], or code fragment level [31, 79, 134], assume that the attacker can manipulate the input to exploit the vulnerability in the regex. However, this assumption may not always be applicable. For instance, consider a library that only reads and parses a configuration file on a server, which is not controlled by an attacker. In such cases, the likelihood of an attacker being able to influence the input and exploit the ReDoS vulnerability is significantly reduced, if not eliminated.

Attacker's Input Reaches Super-Linear Regex (34 studies): This condition necessitates that the attacker's input be matched against a specific type of regex. While all the papers considering threat models at the library and code fragment levels implicitly assume this condition, it is a significant assumption. With the exception of Barlas *et al.* [6], most studies presume this might be achievable in practice. In reality, achieving this could require the attacker to have access to detailed information about the server-side logic, API documentation, or the internal workings of the web service. Staicu *et al.* [120] incorporate web server implementations into their threat model and examine whether the input can trigger the vulnerability. However, this fingerprinting approach can be noisy, leading to both false positives and false negatives.

Use of Backtracking or Slow Regex Engine (33 studies): This condition assumes the attacker has in-depth knowledge of the specific regex engine employed by the server. On one hand, it might be feasible to remotely determine the type of engine in use (*e.g.*, PCRE, RE2), but pinpointing the exact version with its specific weaknesses can be quite challenging. On the other hand, assuming a slow or outdated engine might lead to inaccurate results, making the attack scenario unrealistic in many real-world situations. Except for Barlas *et al.* [6] and Turoňová *et al.* [134], all the studies make this assumption as part of their threat model.

Arbitrary Request Assumption (32 studies): This condition represents a strong assumption that the attacker can repeatedly send requests with large input lengths without interference from other security mechanisms. Most studies, with the exceptions of [79, 97, 113], either disregard restrictions on request size and frequency or necessitate large inputs to trigger vulnerabilities. For context, the maximum size of HTTP header requests and responses in `nginx` and `Apache Tomcat` [16, 41] is 8K, which is significantly smaller than the input lengths assumed in most works. For instance, Davis *et al.* [36] require input lengths of 100K-1M to trigger a 10-second slowdown, which would be impractical in real-world system settings. Furthermore, most studies fail to consider potential mitigation strategies employed by real-world systems, such as rate limiting, intrusion detection systems, or firewalls, which can significantly impede attacks. In realistic scenarios, attackers might face various system-imposed limitations, including bandwidth constraints or restrictions on the number of requests [110] they can send. As a result, the majority of studies, except for [79, 113, 122], assume in their threat models that attackers can perform requests unhindered, which may not reflect real-world conditions.

While the inclusion of a realistic threat model is essential for evaluating security findings accurately, many existing ReDoS studies make unrealistic, strong assumptions about attacker capabilities and system configurations.

3.3 ReDoS Definitions

Most of the considered papers define a ReDoS vulnerability in terms of the slowdown that can be caused directly, with a limited number of characters. This variation is summarized in Table 3. Notably, there is no consensus on what constitutes a vulnerability. For instance, some studies establish thresholds for the number of allowed operations during matching that differ significantly. Liu *et al.* [79] and Wang *et al.* [138] use a threshold of 10^5 instructions, while Shen *et al.* [113] and Siddiq *et al.* [115] consider 10^8 instructions, and Parolini *et al.* [97] set it at 10^{10} instructions. Adding to the confusion, Sung *et al.* [123] define vulnerabilities at 10^6 and 10^7 instructions. It remains unclear why these values are deemed appropriate, as heavy CPU loads do not always indicate the presence of a vulnerability.

Table 3: Summary of ReDoS vulnerability definitions in reviewed studies. The literature uses inconsistent criteria, with over half (19/29) defining ReDoS based on slowdown. ◐ denotes incomplete details; ● indicates use of heuristics; – means the study did not use or report that criterion.

Paper	Slowdown	Number of Instructions	Heuristics
Su *et al.* [122]	–	>10000	–
Davis *et al.* [31]	–	–	●
Turoňová *et al.* [134]	10s, 100s	–	–
Liu *et al.* [79]	–	10^5	–
Staicu *et al.* [120]	5s	–	–
Davis *et al.* [36]	10s	–	–
Siddiq *et al.* [116]	1s	–	–
Shen *et al.* [113]	–	10^8	–
Barlas *et al.* [6]	1s	–	–
Siddiq *et al.* [115]	1s	10^8	–
Davis *et al.* [37]	10s	–	–
Kluban *et al.* [70]	–	–	●
Bhuiyan *et al.* [9]	2s	–	–
Wang *et al.* [138]	–	10^5	–
McLaughlin *et al.* [84]	10s	–	–
Li *et al.* [73]	1s	–	–
Parolini *et al.* [97]	–	10^{10}	–
Wüstholz *et al.* [141]	◐	–	–
Petsios *et al.* [100]	◐	–	–
Kluban *et al.* [71]	–	–	●
Demoulin *et al.* [40]	◐	–	–
Chida *et al.* [15]	–	–	●
Bai *et al.* [5]	$\mu + 3\sigma$	–	–
Li *et al.* [74]	10s	–	–
Li *et al.* [75]	◐	–	–
Hassan *et al.* [63]	–	–	●
Davis *et al.* [39]	1s	–	–
Davis *et al.* [38]	◐	–	–
Turoňová *et al.* [133]	10s, 100s	–	–

Moreover, the inconsistency extends to the measurement of introduced slowdowns. Various papers use different metrics to quantify slowdowns, further complicating the landscape and hindering comparability across studies. Turoňová *et al.* [134] measure slowdowns in 10*s* and 100*s*, Staicu *et al.* [120] in 5*s*, Davis *et al.* [36, 37] in 10*s*, Siddiq *et al.* [116], Li *et al.* [73] and Barlas *et al.* [6] in 1*s*, Bhuiyan *et al.* [9] in 2*s*. Meanwhile, McLaughlin *et al.* [84] use 10*s*, and Bai *et al.* [5] define a problematic slowdown as the average response time (μ) plus three times the standard deviation (3σ), to account for significant deviations from the norm. These excessively high thresholds likely miss many potential vulnerabilities that could cause significant issues under real-world conditions. In our experiments, we saw that under one-second slowdowns can be weaponized against a target server, with modest attacker resources [10]. Other papers rely on heuristics to define a vulnerability. For example, Davis *et al.* [31] utilize safe-regex [34], which uses static analysis rules to determine whether a regex is vulnerable. These tools are unsound and report both false positives and negatives.

The lack of consensus on vulnerability thresholds, slowdown metrics, and measurement methodologies highlights significant

Table 4: Summary of evaluation methods used in the reviewed studies. Most works (33/34) measure the success of a ReDoS attack from the perspective of the attacker, rather than from its impact on benign users (*i.e.*, Quality of Service (QoS) degradation). ● indicates that the criterion or platform was explicitly considered; ◐ denotes incomplete details; – means the study did not use or report that criterion.

Paper	Attack Simulation	Measure QoS Degradation	JavaScript	Java	Python	Ruby	PHP	Go	Perl	Rust	C#
Su *et al.* [122]	–	–	●	●	–	–	●	●	●	●	●
Davis *et al.* [31]	–	–	●	–	–	–	–	–	–	–	–
Turoňová *et al.* [134]	–	–	●	●	●	●	●	–	●	–	●
Liu *et al.* [79]	–	–	●	●	●	●	●	–	–	–	–
Staicu *et al.* [120]	◐	–	●	–	–	–	–	–	–	–	–
Davis *et al.* [36]	–	–	●	–	●	–	–	–	–	–	–
Siddiq *et al.* [116]	–	–	–	●	–	–	–	–	–	–	–
Shen *et al.* [113]	–	–	–	●	–	–	–	–	–	–	–
Barlas *et al.* [6]	◐	–	●	–	–	–	–	–	–	–	–
Siddiq *et al.* [115]	–	–	–	●	–	–	–	–	–	–	–
Davis *et al.* [37]	–	–	●	●	●	●	●	●	●	●	–
Kluban *et al.* [70]	–	–	●	–	–	–	–	–	–	–	–
Bhuiyan *et al.* [9]	–	–	●	–	–	–	–	–	–	–	–
Noller *et al.* [92]	–	–	–	●	–	–	–	–	–	–	–
Wang *et al.* [138]	–	–	●	●	●	–	–	–	–	–	●
Rathnayake *et al.* [106]	–	–	–	–	●	–	–	–	–	–	–
McLaughlin *et al.* [84]	–	–	●	–	–	–	–	–	–	–	–
Li *et al.* [73]	–	–	–	●	–	–	–	–	–	–	–
Parolini *et al.* [97]	–	–	–	●	–	–	–	–	–	–	–
Kirrage *et al.* [68]	–	–	–	●	–	–	–	–	–	–	–
Wüstholz *et al.* [141]	◐	–	–	●	–	–	–	–	–	–	–
Petsios *et al.* [100]	–	–	–	–	–	●	–	–	–	–	–
Kluban *et al.* [71]	–	–	●	–	–	–	–	–	–	–	–
Demoulin *et al.* [40]	●	–	●	–	–	–	–	–	–	–	–
Chida *et al.* [15]	–	–	●	–	●	–	–	–	–	–	–
Bai *et al.* [5]	●	●	●	–	–	–	–	–	–	–	–
Li *et al.* [74]	–	–	–	●	–	–	–	–	–	–	–
Davis *et al.* [32]	–	–	●	●	●	●	●	●	●	●	–
Li *et al.* [75]	–	–	–	●	–	–	–	–	–	–	–
Hassan *et al.* [63]	–	–	●	●	●	●	●	–	●	–	●
Davis *et al.* [39]	◐	–	●	–	–	–	–	–	–	–	–
Tandon *et al.* [125]	●	–	–	–	–	–	●	–	–	–	–
Weideman *et al.* [139]	–	–	–	●	–	–	–	–	–	–	–
Turoňová *et al.* [133]	–	–	–	–	–	–	–	–	–	–	●

inconsistencies in ReDoS research, underscoring the need for standardized definitions and evaluation criteria.

3.4 Evaluations

We carefully studied the evaluations of the papers in our dataset, as shown in Table 4. Our analysis reveals two key aspects regarding ReDoS research:

Language: Regex matching is comparatively slow in programming languages like JavaScript, Java, Python, and Ruby [37], leading to a significant amount of research focusing on detecting and defending ReDoS attacks in these languages. Reflecting this emphasis, 19 (54%) papers address JavaScript, 18 (51%) address Java, and 9 (25%) address Python, with JavaScript being particularly prominent.

Evaluation Methodology: Out of the reviewed studies, only 7 (18%), such as Demoulin *et al.* [40], Tandon *et al.* [125], and Staicu *et al.* [120], performed an end-to-end simulation to evaluate their proposed attack or defense methodologies. Moreover, only Bai *et al.* [5] measured the system's performance degradation during an attack as perceived by benign users. The remaining 29 studies (82%) rely solely on theoretical or localized analysis, which might not capture real-world attack scenarios and defenses accurately, potentially overestimating the prevalence of ReDoS vulnerabilities.

The lack of comprehensive end-to-end evaluations in current research hampers the ability to accurately gauge the real-world effectiveness and impact of ReDoS attacks and defenses.

4 ReDoS Mitigations in Regex Engines

In §3, we discuss many papers that measure ReDoS or propose defenses. They often assume that the matching of a regex engine can be modeled as a backtracking search (§3.2). However, it is unclear whether this assumption holds in practice, particularly in the context of modern regex engines.

In this section, we examine the regex engines of major programming languages to answer two questions: (1) *Do the measurements and assumptions in the academic literature accurately reflect the latest versions of these engines?* and (2) *What is the impact and adoption of ReDoS defenses proposed by researchers in real-world regex engines?*

We answer these questions in two ways. First, a systematic analysis was conducted on regex engines in nine popular programming languages (§4.1). This analysis drew upon first-party sources such as source code, issue trackers, language documentation, blog posts from regex engine maintainers, and CVE reports. We also examined academic literature that describes regex engines. Second, we measured whether super-linear regexes improved performance in the latest versions of these engines (§4.2).

4.1 Regex Engine Analysis

In this section, we begin by outlining the selection criteria used to choose the regex engines for our analysis. Then, for each engine, we examine the type and details of the ReDoS defense(s) implemented (if any), the update context of the defense (including the year of introduction), and the required lines of code (LOC) to implement it. Additionally, we assess whether the defense is enabled by default or requires developer intervention. We summarize our results on evaluating ReDoS defenses in modern regex engines in Table 5.

Selection Criteria: Many modern programming languages have multiple implementations, runtime environments, engines, interpreters, or compilers that might use different regex engines. For instance, JavaScript has multiple engines for executing the source code, which can be used in different environments, such as V8 in Node.js and SpiderMonkey in Mozilla Firefox. To address this complexity, we applied two key criteria when selecting a specific regex engine for languages with multiple options. First, we prioritized engines with substantial real-world adoption, as these are the most likely targets of ReDoS attacks in practice. Second, in cases where usage data is sparse or fragmented, we selected reference implementations that are officially endorsed and openly maintained, enabling thorough analysis of ReDoS defenses. An overview of our regex engine selections for this study and their rationale can be found in Appendix D.1.

4.1.1 *JavaScript (Node.js—V8).*

Original Algorithm: The main JavaScript engine, V8, previously used a backtracking Spencer-style algorithm. It was highly optimized for common-case regexes [61]. It uses an explicit automaton representation, optimizing it and then rendering it as native machine code for execution [26].

ReDoS Defense: In 2021, responding to ReDoS concerns, the V8 developers implemented an extra Thompson-style non-backtracking

Table 5: Summary of ReDoS defenses in the implementations of major programming languages.

Language (*Implementation*)	Nature of the Defense	Deployment Year	LOC to Implement	On by Default?	Current ReDoS-Safeness
JavaScript (*Node.js—V8* [44, 102])	A Non-backtracking Engine: Thompson-style	2021	1558 (10.6%)	✗	Partial
Ruby (*MRI/CRuby* [17])	Memoization, Resource Capping: Time-based	2022	1100 (4.7%)	✓	Partial
C# (*.NET* [87])	Resource Capping: Time-based; Brzozowski Derivatives Engine	2012, 2022	5417 (34.7%)	✗	Partial
Java (*OpenJDK* [95])	Caching	2016	1712 (35.5%)	✓	Partial
PHP (*Zend Engine* [59, 60])	Resource Capping: Counter-based	–	–	✓	Partial
Perl (*perl5* [98])	Caching, Resource Capping: Counter-based	–	–	✓	Partial
Rust (*rustc* [50])	A Non-backtracking Engine: Thompson-style	–	–	✓	Safe
Go (*gc* [128])	A Non-backtracking Engine: Thompson-style	–	–	✓	Safe
Python (*CPython* [46])	–	–	–	✗	Not Safe

regex engine [11]. The new engine guarantees linear-time complexity for all supported regexes, but as a trade-off, it does not support some E-regex features. All regexes with unsupported E-regex features still have to use the backtracking engine. This update was introduced in V8 version 8.8, but in Node.js, it remains an experimental feature that is not enabled by default and must be activated via a feature flag. Thus, some Node.js applications may not yet benefit from the new non-backtracking regex engine.

Engineering Effort: The new engine is 1558 LOC. The original was 14741 LOC (excluding platform-specific assembly code). The effort was ~10% of the original.

4.1.2 *Ruby (MRI/CRuby).*

Original Algorithm: Ruby's default regex engine is based on Onigmo [124], a fork of the Oniguruma [72] library, which is a Spencer-style backtracking regex engine.

ReDoS Defense: Until Ruby 3.2 (2022), Ruby's port of Onigmo did not have built-in mitigations against ReDoS. In Ruby 3.2, the developers added two ReDoS defenses [90, 114]. First, based on a community proposal,[2] Ruby implemented Davis *et al.*'s [38] memoization technique for Spencer's algorithm as a defense against ReDoS attacks. Interestingly, the Ruby developers used full memoization rather than selective memoization despite the higher space complexity. Ruby does not apply memoization when unsupported E-regex features are encountered, instead offering a timeout mechanism similar to C# as the second kind of defense.[3] The memoization defense is on by default.

Engineering Effort: The relevant pull requests added a total of 1100 LOC, compared to the Ruby 3.1.6 engine's footprint of 23384 LOC. The ratio is 5%.

4.1.3 *C# (.NET).*

Original Algorithm: The first two regex engines in .NET used backtracking algorithms [20, 21]. One was a simple opcode interpreter. The other used an optimized machine code representation like V8's [19, 21, 62, 132].

ReDoS Defense: In .NET v4.5 (2012), .NET introduced a timeout mechanism to prevent excessive backtracking [23]. The timeout is checked after at most $O(n)$ steps, where n is the length of the input, balancing fidelity with overhead [22, 131]. In .NET 7 (2022), Microsoft added a non-backtracking regex engine [131]. This engine is based on Moseley *et al.* [89] and Saarikivi *et al.* [109], with its core algorithm using Brzozowski derivatives and guaranteeing linear-time matches for supported regexes. The new engine raises an error upon encountering unsupported E-regex features, so the developers must use the backtracking engine in such cases. However, both the timeout mechanism and the non-backtracking engine are opt-in features.

Engineering Effort: The new engine comprises 5417 LOC. The existing .NET codebase for regexes was 15597 LOC (including multiple engines). Hence, the new engine expands the existing regex engine codebase by 35%. Microsoft researchers also created two research papers.

4.1.4 *Java (OpenJDK).*

Original Algorithm: OpenJDK Java's previous regex engine implemented an NFA-based Spencer-style backtracking algorithm for regex matching [96].

ReDoS Defense: As of Java 22 (2024), OpenJDK Java's regex engine does not document defenses against ReDoS. Despite the absence of official documentation, in the OpenJDK source code repository and issue tracker, we identified that some performance updates related to ReDoS issues were introduced in Java 9 (2016).[4] The OpenJDK maintainers introduced a bounded caching mechanism, improving problematic behavior for some ReDoS scenarios. This ReDoS defense is enabled by default and affects all OpenJDK Java versions ≥9.

Engineering Effort: Before the update, the regex-related codebase consisted of 4823 LOC. The memoization patch added 1712 lines (including empty lines and docstrings) to the Java 9 source code, which accounted for a 35% expansion.

4.1.5 *Other ReDoS-Vulnerable Engines.* PHP (Zend Engine), Perl (perl5), and Python (CPython) are also known to be using Spencer-style backtracking regex engines [37, 51], which are vulnerable to ReDoS attacks. According to Davis *et al.* [38], PHP [101] and Perl [81] utilize "counter-based caps" to prevent catastrophic backtracking as an alternative to C# and Ruby's "time-based caps". Python has neither built-in engine optimizations against ReDoS vulnerabilities nor a native timeout-like mechanism for regex evaluations. We have not found any new documented updates or defenses against ReDoS for these languages since the analysis of Davis *et al.* [38] in 2021.

4.1.6 *Other ReDoS-Safe Engines.* Two major programming languages, Rust (rustc, via its official regex crate) [48] and Go (gc) [129], have regex engines that were developed with ReDoS safety in mind. These engines simulate an explicit automaton representation using

[2]See https://bugs.ruby-lang.org/issues/19104.
[3]See https://bugs.ruby-lang.org/issues/17837.
[4]See https://github.com/openjdk/jdk/commit/b45ea89.

Thompson's algorithm [27]. They also do not support many E-regex features. They, therefore, offer linear-time match guarantees in input and automaton length.

Although not included in our analysis, independent third-party regex engines like RE2 (Google) [118] and Hyperscan (Intel) [136, 137] are well-known for ReDoS resilience. RE2, a Thompson-style engine, incorporates DFA state caching optimizations [89]. While generally ReDoS-safe and non-backtracking, RE2 can, in rare cases, exhibit backtracking behavior [38]. Similarly, Hyperscan is another efficient, ReDoS-safe engine, utilizing hardware acceleration alongside Glushkov's NFA construction [54]. Neither engine supports E-regex features that require backtracking.

Overall, these developments in regex engines demonstrate significant progress in mitigating ReDoS risks but also highlight ongoing challenges in balancing the trade-off between safety and usability. Modern regex engines mitigate ReDoS with non-backtracking algorithms like memoization, Thompson-style simulation, or Brzozowski derivatives to assure linear-time matching. However, the benefit comes at the cost of reduced expressiveness. Non-backtracking engines lack support for many E-regex features. Defenses add timeouts or fallbacks for unsupported E-regexes to maintain codebase compatibility.

4.2 Updated Measure of ReDoS Vulnerabilities

As indicated in Table 5, several major regex engines have recently been updated to mitigate or eliminate ReDoS. We, therefore, wondered what proportion of ReDoS vulnerabilities have been addressed by state-of-the-art engines.

We answered this question through measurements on a sample from the Davis *et al.*'s [37] polyglot regex corpus, comprising ∼500K regexes extracted from open-source repositories across eight programming languages. We used the tool vuln-regex-detector [35], developed by Davis *et al.* [36] and based on prior work [106, 113, 139, 141], to identify potentially vulnerable regexes from the corpus. From its predictions, we randomly sampled 500 regexes labeled as exponential-time candidates (out of ∼3K predicted examples) and 500 labeled as polynomial-time candidates (out of ∼100K predicted examples). For each sampled regex, we used the same tool to generate corresponding attack input strings.

Then, we evaluated each regex-input pair on the regex engines described in Table 5, comparing the performance of the engine version before any changes to that of the most recent version. If ReDoS defenses were available but not enabled by default in the latest version—as in the case of JavaScript and C#—we activated them. Following a parallel methodology that has been used in previous work [36, 37], we define a regex's behavior as:

- *Exponential*: The processing time exceeds five seconds with fewer than 500 pumps of an attack string.
- *High-Polynomial*: The processing time exceeds five seconds with 500 or more pumps of an attack string.
- *Low-Polynomial*: The processing time exceeds 0.5 seconds on an attack string but does not meet the exponential or high-polynomial behavior criteria.
- *Linear*: The regex match never times out on the attack string.

Our findings are illustrated in Figure 5. For each engine, we depict the number of regexes that exhibited super-linear behavior (as a fraction of the predicted input) in the pre- and post-ReDoS mitigation versions. Not all regexes were supported in all engines, so the candidate pool size is indicated for each engine. Observe that for the engines with mitigations, there is a substantial decrease in the number of viable super-linear regexes—in other words, these mitigations appear to be effective in addressing the common causes of super-linear behavior. Notably, exponential behavior is resolved in C# and JavaScript but persists in Ruby and Java, while polynomial behavior continues to manifest in JavaScript and Java.

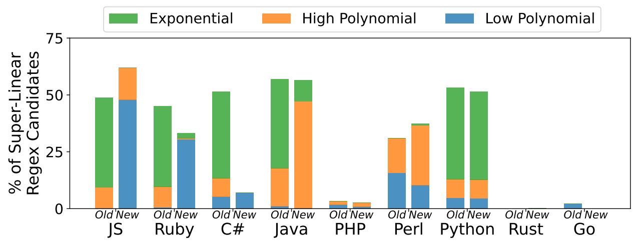


Figure 5: Performance of regex matching in old *vs.* latest engines in different programming languages. The exact engine versions are listed in Appendix D.2.

5 Discussion

In this section, we distill and contextualize the main findings of our study, integrating insights from our literature review (§3) and engineering review (§4). For each idea, we discuss concrete future work directions for the community to explore.

(1) ReDoS is a peculiar vulnerability: Unlike traditional ones like SQL injections, ReDoS stands out as a contextual issue, more similar to micro-architectural attacks, where exploitability and impact are heavily dependent on the underlying system's architecture and deployment conditions. For example, a regex pattern vulnerable to catastrophic backtracking in older versions of Node.js is rendered safe in newer versions due to advancements in regex engines (§4). When detecting ReDoS vulnerabilities, it is not enough to analyze the code of a web application, without considering its deployment configurations. For future static analysis work on ReDoS—particularly those targeting multiple languages such as [63]—we recommend that they consider and model the language-, engine-, and platform-specific details and differences with respect to ReDoS. For example, a super-linear regex might be more serious in a single-threaded web application architecture like Node.js than in inherently parallel ones like Go. For dynamic and hybrid analyses, we recommend a system-level perspective, similar to Shan *et al.* [112], that deploys code under realistic conditions, taking relevant defenses into account.

(2) Strong assumptions about attackers: We find that prior work often makes strong assumptions about attackers (§3.2), such as assuming they can inject arbitrarily large payloads or control the input to any regex. Also, many papers neglect defenses or rely on rules of thumb, like regex matches should not exceed one second (§3.3). We think that future work should challenge these assumptions by performing empirical studies with real systems, including analyses of how many regexes are actually reachable via attacker-controlled inputs. We also need more empirical studies to show the community how modern applications are deployed in the wild—specifically, which configurations they run on and what defenses they employ.

(3) Developers push back: To the best of our knowledge, there has been no public description of a DoS attack that exploited ReDoS, so the risk posed by these vulnerabilities is still unclear, beyond academic measurement studies [6, 120]. Moreover, only a small fraction of the papers we analyzed (§3) discuss such attacks, often in unrealistic setups. We believe that these unclear risks together with problematic security assumptions are the culprits for ReDoS vulnerabilities being a contentious topic among developers. Snyk has stated that ReDoS is the most neglected vulnerability type [77], implying that developers often ignore such findings. Also, engineers at GitHub consider it a low severity vulnerability, "not particularly serious, but easy to create by accident, obscure to understand, and sometimes tricky to fix" [4]. More strident voices argue that ReDoS vulnerabilities are "malicious noise" and just lead to security fatigue [140]. Several ReDoS CVE advisories have been disputed by the maintainers (*e.g.*, CVE-2023-39663, CVE-2022-42969, CVE-2019-11391), indicating disagreement between maintainers and reporters, also reflected by negative comments in GitHub issues (Appendix E).

The blame for this perspective may lie with the cybersecurity academic community. As noted in §3, researchers have performed large-scale measurements of ReDoS as well as produced new and more effective detectors for vulnerable regexes. Their measurement instruments have been integrated into popular security scanners (*e.g.*, the integration of safe-regex into eslint). However, the resulting scanners analyze only the regex, without considering reachability constraints as well as other input constraints. When academics apply these tools, they are likely to filter false positives based on the regex's context. In contrast, practitioners sometimes make much ado about nothing. This suggests the need for empirical studies of academic tool uptake and more developer-friendly releases. More importantly, future work in the field should perform user studies with developers to better understand their challenges related to ReDoS—such as tool usability, security assumptions, and deployment configurations. In this way, we can better align the community's assumptions with those of practitioners.

(4) Engines have evolved significantly: It is encouraging that modern language runtimes changed their implementation in response to prior work on ReDoS (§4). These changes were often driven or informed by researchers (*e.g.*, Ruby's adoption of Davis *et al.*'s defenses [32, 38]). We believe that this is a success story of effective collaboration between researchers and practitioners.

Our study shows that modern regex engines have evolved significantly, incorporating advanced techniques such as Thompson's algorithm, memoization, and resource capping to mitigate ReDoS. However, there are often redundant implementations that are triggered on demand by the engine, when matching a regex. Hence, future work should investigate the deployed fixes to find potential bugs or misbehaving corner cases. We advocate for comparative studies (such as ours in §4.2), where multiple regex engines are evaluated on the same regexes or datasets to systematically assess the strengths and weaknesses of their deployed defenses. The research community should also propose automatic solutions for migrating to updated engines. At the same time, research on ReDoS should take these advancements into account to ensure its findings remain applicable.

(5) Performance bugs vs. security vulnerabilities: Many of our findings may also extend to other algorithmic complexity vulnerabilities. While the threat of weaponizing slow computations is real, in practice, attackers still tend to favor brute-force, volume-based distributed DoS attacks. Researchers should aim to understand why that is the case and, simultaneously, continue to develop defenses against such potential future attacks. We also believe that there are many insufficiently explored, cross-disciplinary opportunities for ideas that aim to bridge the gap between performance, measurement, and security communities. For example, we argue that most performance bugs (*e.g.*, redundant loop traversals [94]) can probably be lifted to a DoS attack when malicious intent is assumed. Thus, future work should further examine the relation between performance bugs and DoS attacks.

6 Conclusion

In this work, we perform a multi-faceted study of the recent academic work on ReDoS and the new engineering realities in the mainstream programming languages, after initial fixes were deployed in most of them. We report on concrete examples of language runtimes that adopted defenses, which appear to be motivated by academic research. When studying the efficacy of these defenses, we find that they mitigate most ReDoS payloads, but they still leave significant room for attackers to maneuver. By surveying the academic work on ReDoS, we find that many papers in this domain use weak definitions and strong threat models, often considering any slowdown caused by regex matching as a security vulnerability. We advocate for future work to consider the recently-deployed fixes in mainstream programming languages and to evaluate ReDoS attacks under realistic deployment conditions (*e.g.*, against a web application deployed in the cloud using reversed proxies and redundancy). We also propose restricting the attackers' capabilities to realistic payload sizes and modest bandwidths that comply with the asymmetric DoS setting. More importantly, future work should study the more general problem of weaponizing slow computation, which the attackers might leverage in the near future.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grant #2135156.

Ethical Considerations

In our judgment, this work presents an acceptable ethical risk-reward tradeoff. Specifically, the literature review (§3) and engineering review (§4) describe and systematize only publicly available resources.

Data Availability

An artifact accompanying this paper is available at https://doi.org/10.5281/zenodo.15515222. It includes: the CVE data analyzed in §2.3.2, the super-linear regex corpus together with the reproduction scripts for the experiments described in §4.2, and the developer-discussion data and GitHub-crawler scripts from Appendix E.

References

[1] Alfred V. Aho. 1980. Pattern Matching in Strings. In *Formal Language Theory*, RONALD V. Book (Ed.). Academic Press, 325–347. doi:10.1016/B978-0-12-115350-2.50016-6

[2] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. 2008. General Algorithms for Testing the Ambiguity of Finite Automata. In *Developments in Language Theory*, Masami Ito and Masafumi Toyama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 108–120.

[3] Inc. Amazon Web Services. 2024. Regex Pattern Set Match Rule Statement - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced. https://docs.aws.amazon.com/waf/latest/developerguide/waf-rule-statement-type-regex-pattern-set-match.html

[4] Kevin Backhouse. 2023. How to fix a ReDoS. https://github.blog/2023-05-09-how-to-fix-a-redos/

[5] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1575–1588. doi:10.1109/SP40001.2021.00077

[6] Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting input sanitization for regex denial of service. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 883–895. doi:10.1145/3510003.3510047

[7] Martin Berglund, Brink Van Der Merwe, and Steyn Van Litsenborgh. 2021. Regular Expressions with Lookahead. *JUCS - Journal of Universal Computer Science* 27, 4 (April 2021), 324–340. doi:10.3897/jucs.66330

[8] Martin Berglund, Brink Van Der Merwe, Bruce Watson, and Nicolaas Weideman. 2017. On the Semantics of Atomic Subgroups in Practical Regular Expressions. In *Implementation and Application of Automata*, Arnaud Carayol and Cyril Nicaud (Eds.). Vol. 10329. Springer International Publishing, Cham, 14–26. doi:10.1007/978-3-319-60134-2_2

[9] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1059–1070. doi:10.1109/ICSE48619.2023.00096

[10] Masudul Hasan Masud Bhuiyan and Cristian-Alexandru Staicu. 2022. A Tale of Frozen Clouds: Quantifying the Impact of Algorithmic Complexity Vulnerabilities in Popular Web Servers. arXiv:2211.11357 [cs.CR] https://arxiv.org/abs/2211.11357

[11] Martin Bidlingmaier. 2021. An Additional Non-Backtracking RegExp Engine. https://v8.dev/blog/non-backtracking-regexp

[12] James Buchanan. 2023. Regex Parsing: Extract Data from Logs. https://newrelic.com/blog/how-to-relic/extracting-log-data-with-regex

[13] Elasticsearch B.V. 2024. Accessing Event Data and Fields. https://www.elastic.co/guide/en/logstash/current/event-dependent-configuration.html#conditionals.

[14] Agnishom Chattopadhyay, Angela W. Li, and Konstantinos Mamouras. 2025. Verified and Efficient Matching of Regular Expressions with Lookaround. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 198–213. doi:10.1145/3703595.3705884

[15] Nariyoshi Chida and Tachio Terauchi. 2022. Repairing DoS Vulnerability of Real-World Regexes. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2060–2077. doi:10.1109/SP46214.2022.9833597

[16] NGINX Community. 2024. Module ngx_http_core_module. http://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers

[17] Ruby Community. 2024. About Ruby. https://www.ruby-lang.org/en/about/

[18] Ruby Community. 2024. Ruby/Ruby. https://github.com/ruby/ruby

[19] .NET Contributors. 2021. Compilation and Reuse in Regular Expressions. https://learn.microsoft.com/en-us/dotnet/standard/base-types/compilation-and-reuse-in-regular-expressions

[20] .NET Contributors. 2021. Regular Expression Behavior. https://learn.microsoft.com/en-us/dotnet/standard/base-types/details-of-regular-expression-behavior

[21] .NET Contributors. 2023. Backtracking in .NET Regular Expressions. https://learn.microsoft.com/en-us/dotnet/standard/base-types/backtracking-in-regular-expressions

[22] .NET Contributors. 2023. Best Practices for Regular Expressions in .NET. https://learn.microsoft.com/en-us/dotnet/standard/base-types/best-practices-regex

[23] .NET Contributors. 2023. What's New in .NET Framework. https://learn.microsoft.com/en-us/dotnet/framework/whats-new/#whats-new-in-net-framework-45

[24] .NET Contributors. 2024. C# Language Specification. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction

[25] Oracle Corporation. 2024. Openjdk/Jdk. https://github.com/openjdk/jdk

[26] Erik Corry, Christian Plesner Hansen, and Lasse Reichstein Holst Nielsen. 2009. Irregexp, Google Chrome's New Regexp Implementation. https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html

[27] Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast. https://web.archive.org/web/20240529192558/https://swtch.com/~rsc/regexp/regexp1.html

[28] Scott Crosby. 2003. Denial of Service through Regular Expressions. USENIX Association, Washington, D.C.

[29] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association, Washington, D.C.

[30] James Davis. 2020. The Regular Expression Denial of Service (ReDoS) Cheat-Sheet. https://levelup.gitconnected.com/the-regular-expression-denial-of-service-redos-cheat-sheet-a78d0ed7d865

[31] James Davis, Gregor Kildow, and Dongyoon Lee. 2017. The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture. In *Proceedings of the 10th European Workshop on Systems Security* (Belgrade, Serbia) *(EuroSec'17)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3065913.3065916

[32] James C. Davis. 2019. Rethinking Regex engines to address ReDoS. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1256–1258. doi:10.1145/3338906.3342509

[33] James Collins Davis. 2020. *On the impact and defeat of regular expression denial of service*. PhD Thesis. Virginia Tech.

[34] James C. Davis. 2024. Davisjam/safe-regex. https://github.com/davisjam/safe-regex

[35] James C. Davis. 2024. Davisjam/Vuln-Regex-Detector. https://github.com/davisjam/vuln-regex-detector

[36] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 246–256. doi:10.1145/3236024.3236027

[37] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 443–454. doi:10.1145/3338906.3338909

[38] James C. Davis, Francisco Servant, and Dongyoon Lee. 2021. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17. doi:10.1109/SP40001.2021.00032

[39] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 343–359.

[40] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. 2019. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 693–708.

[41] Apache Software Foundation. 2018. core - Apache HTTP Server Version 2.2. https://httpd.apache.org/docs/2.2/mod/core.html#limitrequestfieldsize

[42] .NET Foundation. 2024. Dotnet/Runtime. https://github.com/dotnet/runtime

[43] OWASP Foundation. 2021. OWASP Top 10. https://owasp.org/Top10/.

[44] OpenJS Foundation. 2024. About Node.js. https://nodejs.org/en/about

[45] OpenJS Foundation. 2024. Nodejs/Node. https://github.com/nodejs/node

[46] Python Software Foundation. 2024. Python Developer's Guide. https://devguide.python.org/

[47] Python Software Foundation. 2024. Python/Cpython. https://github.com/python/cpython

[48] Rust Foundation. 2024. Rust-Lang/Regex. https://github.com/rust-lang/regex

[49] Rust Foundation. 2024. Rust-Lang/Rust. https://github.com/rust-lang/rust

[50] Rust Foundation. 2024. What is rustc? https://doc.rust-lang.org/stable/rustc/.

[51] Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions* (3 ed.). O'Reilly Media, Sebastopol, CA.

[52] Hiroya Fujinami and Ichiro Hasuo. 2024. Efficient Matching with Memoization for Regexes with Look-around and Atomic Grouping. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Vol. 14577. Springer Nature Switzerland, Cham, 90–118. doi:10.1007/978-3-031-57267-8_4

[53] Virgil D. Gligor. 1983. A Note on the Denial-of-Service Problem. In *1983 IEEE Symposium on Security and Privacy*. 139–139. doi:10.1109/SP.1983.10004 ISSN:

1540-7993.

[54] V M Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (Oct. 1961), 1–53. doi:10.1070/RM1961v016n05ABEH004112

[55] Jan Goyvaerts. 2019. Preventing Regular Expression Denial of Service (ReDoS). https://www.regular-expressions.info/redos.html

[56] Jan Goyvaerts. 2021. Runaway Regular Expressions: Catastrophic Backtracking. http://www.regular-expressions.info/catastrophic.html

[57] John Graham-Cumming. 2019. Details of the Cloudflare outage on July 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019

[58] PHP Group. 2024. Php/Php-Src. https://github.com/php/php-src

[59] PHP Documentation Group. 2024. PHP: General Information. https://www.php.net/manual/en/faq.general.php

[60] PHP Documentation Group. 2024. PHP: History of PHP and Related Projects. https://www.php.net/manual/en/history.php.

[61] Jakob Gruber. 2017. Speeding up V8 Regular Expressions \cdot V8. https://v8.dev/blog/speeding-up-regular-expressions

[62] David Gutierrez. 2004. Regular Expression Performance. https://learn.microsoft.com/en-us/archive/blogs/bclteam/regular-expression-performance-david-gutierrez

[63] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. 2023. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1238–1255. doi:10.1109/SP46215.2023.10179442

[64] Philip Hazel. 1997. PCRE - Perl Compatible Regular Expressions.

[65] Lukáš Holík, Juraj Síč, Lenka Turoňová, and Tomáš Vojnar. 2023. Fast Matching of Regular Patterns with Synchronizing Counting. In *Foundations of Software Science and Computation Structures*, Orna Kupferman and Pawel Sobocinski (Eds.). Springer Nature Switzerland, Cham, 392–412. doi:10.1007/978-3-031-30829-1_19

[66] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation* (3rd ed ed.). Pearson/Addison Wesley, Boston.

[67] Tom Jenkinson. 2024. Tjenkinson/redos-detector. https://github.com/tjenkinson/redos-detector

[68] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security*, Javier Lopez, Xinyi Huang, and Ravi Sandhu (Eds.). Springer, Berlin, Heidelberg, 135–148. doi:10.1007/978-3-642-38631-2_11

[69] S. C. Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, C. E. Shannon and J. McCarthy (Eds.). Princeton University Press, Princeton, 3–42. doi:doi:10.1515/9781400882618-002

[70] Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2022. On Measuring Vulnerable JavaScript Functions in the Wild. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 917–930. doi:10.1145/3488932.3497769

[71] Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2024. On Detecting and Measuring Exploitable JavaScript Functions in Real-world Applications. *ACM Transactions on Privacy and Security* 27, 1, Article 8 (Feb. 2024), 37 pages. doi:10.1145/3630253

[72] K. Kosako. 2024. Kkos/Oniguruma. https://github.com/kkos/oniguruma

[73] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. 2021. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3847–3864.

[74] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. 2022. RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4183–4200.

[75] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. 2021. FlashRegex: deducing anti-ReDoS regexes from examples. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 659–671. doi:10.1145/3324884.3416556

[76] Snyk Limited. 2019. ReDoS vulnerabilities in npm spikes by 143% and XSS continues to grow. https://snyk.io/blog/redos-vulnerabilities-in-npm-spikes-by-143-and-xss-continues-to-grow/

[77] Snyk Limited. 2023. State of Open Source Security. https://snyk.io/reports/open-source-security/

[78] Snyk Limited. 2024. ReDoS Tutorials & Examples. https://learn.snyk.io/lesson/redos/

[79] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1468–1484. doi:10.1109/SP40001.2021.00062

[80] Doyensec LLC. 2024. Doyensec/regexploit. https://github.com/doyensec/regexploit

[81] Joerg Ludwig. 2009. *Complex regular subexpression recursion limit.* https://www.perlmonks.org/?node_id=810857

[82] Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proc. ACM Program. Lang.* 8, POPL, Article 92 (Jan. 2024), 31 pages. doi:10.1145/3632934

[83] Georgios Mantas, Natalia Stakhanova, Hugo Gonzalez, Hossein Hadian Jazi, and Ali A. Ghorbani. 2015. Application-layer denial of service attacks: taxonomy and survey. *International Journal of Information and Computer Security* 7, 2/3/4 (Nov. 2015), 216–239. doi:10.1504/IJICS.2015.073028

[84] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. 2022. Regulator: Dynamic Analysis to Detect ReDoS. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4219–4235.

[85] R. McNaughton and H. Yamada. 1960. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers* EC-9, 1 (March 1960), 39–47. doi:10.1109/TEC.1960.5221603

[86] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. 2018. Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 393–410.

[87] Microsoft. 2024. What Is .NET? https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

[88] MITRE. 2021. CWE-1333: Inefficient Regular Expression Complexity. https://cwe.mitre.org/data/definitions/1333.html

[89] Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proc. ACM Program. Lang.* 7, PLDI, Article 148 (June 2023), 24 pages. doi:10.1145/3591262

[90] Yui Naruse. 2022. Ruby 3.2.0 Released. https://www.ruby-lang.org/en/news/2022/12/25/ruby-3-2-0-released/.

[91] Inc. New Relic. 2024. State of the Java Ecosystem Report. https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem

[92] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 322–332. doi:10.1145/3213846.3213868

[93] Nsrav. 2024. Denial of Service. https://owasp.org/www-community/attacks/Denial_of_Service

[94] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 369–378. doi:10.1145/2737924.2737966

[95] Oracle Corporation. 2024. OpenJDK. https://openjdk.org/

[96] Oracle Corporation. 2024. Pattern (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

[97] Francesco Parolini and Antoine Miné. 2023. Sound static analysis of regular expressions for vulnerabilities to denial of service attacks. *Science of Computer Programming* 229, Article 102960 (July 2023), 48 pages. doi:10.1016/j.scico.2023.102960

[98] Perl.org. 2024. About Perl. https://www.perl.org/about.html

[99] Perl.org. 2024. Perl/Perl5. https://github.com/perl/perl5

[100] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. doi:10.1145/3133956.3134073

[101] PHP Documentation Group. 2024. PHP: Runtime Configuration - Manual. https://www.php.net/manual/en/pcre.configuration.php.

[102] V8 Project. 2024. V8 JavaScript Engine. https://v8.dev/

[103] V8 Project. 2024. V8/v8. https://chromium.googlesource.com/v8/v8.git

[104] Q-Success. 2024. Distribution of Web Servers among Websites That Use JavaScript. https://w3techs.com/technologies/segmentation/pl-js/web_server

[105] M. O. Rabin and D. Scott. 1959. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, 2 (April 1959), 114–125. doi:10.1147/rd.32.0114

[106] Asiri Rathnayake and Hayo Thielecke. 2017. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics (Extended). arXiv:1405.7058 [cs.PL] https://arxiv.org/abs/1405.7058

[107] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington) *(LISA '99)*. USENIX Association, USA, 229–238.

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

[109] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems*,

Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 372–378. doi:10.1007/978-3-030-17462-0_24

[110] Inc. Salesforce. 2024. HTTP Routing. https://devcenter.heroku.com/articles/http-routing

[111] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1974–1993. doi:10.1109/SP54263.2024.00137

[112] Huasong Shan, Qingyang Wang, and Calton Pu. 2017. Tail Attacks on Web Applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1725–1739. doi:10.1145/3133956.3133968

[113] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 225–235. doi:10.1145/3238147.3238159

[114] Victor Shepelev. 2023. Ruby 3.2 Changes. https://rubyreferences.github.io/rubychanges/3.2.html#regexp-redos-vulnerability-prevention.

[115] Mohammed Latif Siddiq, Jiahao Zhang, Lindsay Roney, and Joanna C. S. Santos. 2024. Re(gEx|DoS)Eval: Evaluating Generated Regular Expressions and their Proneness to DoS Attacks. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 52–56. doi:10.1145/3639476.3639757

[116] Mohammed Latif Siddiq, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. 2024. Understanding Regular Expression Denial of Service (ReDoS): Insights from LLM-Generated Regexes and Developer Forums. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension* (Lisbon, Portugal) *(ICPC '24)*. Association for Computing Machinery, New York, NY, USA, 190–201. doi:10.1145/3643916.3644424

[117] Randy Smith, Cristian Estan, and Somesh Jha. 2008. XFA: Faster Signature Matching with Extended Automata. In *2008 IEEE Symposium on Security and Privacy (SP)*. IEEE, 187–201. doi:10.1109/SP.2008.14

[118] Google Open Source. 2024. Google/Re2. https://github.com/google/re2

[119] Henry Spencer. 1994. A Regular-Expression Matcher. In *Software Solutions in C*. Academic Press Professional, Inc., USA, 35–71.

[120] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 361–376.

[121] Stack Exchange Network Status. 2016. Outage Postmortem - July 20, 2016. https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016

[122] Weihao Su, Hong Huang, Rongchen Li, Haiming Chen, and Tingjian Ge. 2024. Towards an Effective Method of ReDoS Detection for Non-backtracking Engines. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 271–288.

[123] Sicheol Sung, Hyunjoon Cheon, and Yo-Sub Han. 2022. How to Settle the ReDoS Problem: Back to the Classical Automata Theory. In *Implementation and Application of Automata*, Pascal Caron and Ludovic Mignot (Eds.). Vol. 13266. Springer International Publishing, 34–49. doi:10.1007/978-3-031-07469-1_3

[124] K. Takata. 2024. K-Takata/onigmo. https://github.com/k-takata/onigmo

[125] Rajat Tandon, Haoda Wang, Nicolaas Weideman, Shushan Arakelyan, Genevieve Bartlett, Christophe Hauser, and Jelena Mirkovic. 2023. Leader: Defense Against Exploit-Based Denial-of-Service Attacks on Web Applications. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (Hong Kong, China) *(RAID '23)*. Association for Computing Machinery, New York, NY, USA, 744–758. doi:10.1145/3607199.3607238

[126] Cisco Talos Detection Response Team. 2024. Regex - Snort 3 Rule Writing Guide. https://docs.snort.org/rules/options/payload/regex

[127] Go Team. 2024. Golang/Go. https://github.com/golang/go

[128] Go Team. 2024. Introduction to the Go Compiler. https://go.dev/src/cmd/compile/README

[129] Go Team. 2024. Regexp Package - Go. https://pkg.go.dev/regexp.

[130] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. doi:10.1145/363347.363387

[131] Stephen Toub. 2022. Regular Expression Improvements in .NET 7. http://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7

[132] Stephen Toub. 2023. Performance Improvements in .NET 8. https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/#regex

[133] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex matching with counting-set automata. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 218 (Nov. 2020), 30 pages. doi:10.1145/3428286

[134] Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2022. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4165–4182.

[135] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL, Article 1 (Jan. 2025), 32 pages. doi:10.1145/3704837

[136] Xiang Wang. 2017. Introduction to Hyperscan. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html

[137] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648.

[138] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, and Wei Huo. 2023. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2427–2443. doi:10.1109/SP46215.2023.10179328

[139] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. 2016. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *Implementation and Application of Automata (Lecture Notes in Computer Science)*, Yo-Sub Han and Kai Salomaa (Eds.). Springer International Publishing, 322–334. doi:10.1007/978-3-319-40946-7_27

[140] William Woodruff. 2022. ReDoS "Vulnerabilities" and Misaligned Incentives. https://news.ycombinator.com/item?id=34161221

[141] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Vol. 10206. Springer Berlin Heidelberg, 3–20. doi:10.1007/978-3-662-54580-5_1

Outline of the Appendices

The appendices contain the following material:

- Reference table of regex features and notation (Appendix A).
- Methodology on how we studied ReDoS in practice (Appendix B).
- Information regarding the exclusion of works from the literature review (Appendix C).
- Additional details of the engineering review, including engine selection and version information (Appendix D).
- A look at the developer discussions about ReDoS vulnerabilities (Appendix E).

A Regex Feature Reference

Table 6 provides a comprehensive reference of regex features and notation, summarized from the PCRE2 specification [64]. The table categorizes regex features into two groups: K-regexes and E-regexes. E-regexes are also subcategorized. Each feature is associated with its syntactic notation.

B ReDoS in Practice

We conducted a comprehensive analysis of NVD CVE data from 2014 to 2023 in order to quantify ReDoS vulnerabilities in practice. As part of this effort, we utilized the CWE field within the CVE reports. For each OWASP Top 10 category, we counted the number of CVEs that included a CWE entry associated with that specific OWASP category.

To identify ReDoS CVEs, we employed a systematic two-stage keyword search:

- **Stage one keywords:** `regex`, `regular expression`, `regexp`
- **Stage two keywords:** `algorithm`, `backtrack`, `uncontrolled`, `repetition`, `repeat`, `infinite`, `denial of service`, `dos`, `infinite loop`, `algorithmic complexity`

Table 6: Full set of features and notation of regexes, compiled from [64]. R denotes a sub-pattern.

Abbreviation	Feature	Notation
K-regexes		
CAT	X followed by Y	R_1R_2
KLE	Zero-or-more repetition	$R*$
OR	Logical OR	$R_1 \vert R_2$
E-regexes		
Capture Groups		
CG	Capture group	(R)
NCG	Non-capture group	$(?:R)$
PNG	Named capture group	$(?\text{<name>}R)$
Quantifiers		
ADD	One-or-more repetition	$R+$
QST	Zero-or-one repetition	$R?$
DBB	Range repetition	$R\{m,n\}$
LWB	At-least-m repetition	$R\{m,\}$
SNG	Exactly-n repetition	$R\{n\}$
Character Classes		
CCC	Custom character class	$[\text{aeiou}]$
RNG	Character range	$[\text{a} - \text{z}]$
NCCC	Negated class	$[\hat{}\text{aeiou}]$
ANY	Any character	.
WSP	Whitespace	\s
DEC	Numeric	\d
WRD	Word	\w
NWSP	Non-whitespace	\S
NDEC	Non-numeric	\D
NWRD	Non-word	\W
VWSP	Vertical space	\v
Zero-width Assertions		
STR	Start-of-string/line	$\hat{}R$, $\backslash AR$
END	End-of-string/line	$R\$$, $R\backslash Z$
WNW	Word/non-word boundary	\b
NWNW	Negated WNW boundary	\B
PLA	Positive lookahead	$(?=R)$
NLA	Negative look-ahead	$(?!R)$
PLB	Positive lookbehind	$(?<=R)$
NLB	Negative look-behind	$(?<!R)$
Backreferences		
BKR	Numeric backreference	$(R)\ldots\backslash 1$
BKRN	Named backreference	$(?\text{<name>}R)\ldots\backslash\text{k<name>}$
Backtracking Controls		
LZY	Non-greedy repetition	$R*?$, $R+?$, $R\{m,n\}?$
ATM	Atomic group	$(?>R)$
POS	Possessive quantifier	$R++$, $R*+$

This method effectively identified ReDoS-related CVEs, and our verification of a sample confirmed no false positives.

C Works Excluded From the Literature Review

Some closely related papers were excluded from the literature review because they do not focus on ReDoS-specific vulnerabilities or their evaluation lacks any ReDoS-specific aspects. These exclusions were necessary to ensure that the review remains precise and relevant to the study of ReDoS. For instance, [27, 112, 135] were excluded because they focused on broader DoS vulnerabilities or algorithmic complexity attacks. Other papers were excluded for evaluating performance issues in regex engines without demonstrating their impact on ReDoS vulnerabilities.

Table 7 highlights examples of these excluded works along with the reasons for their exclusion, showcasing the deliberate scope refinement undertaken to focus on ReDoS-specific contributions.

Table 7: Examples of excluded works with explanations.

Work	Reason for Exclusion
Cox [27]	Evaluation does not address ReDoS-specific scenarios.
Varatalu *et al.* [135]	Evaluation does not address ReDoS-specific scenarios.
Chattopadhyay *et al.* [14]	Evaluation does not address ReDoS-specific scenarios.
Mamouras *et al.* [82]	Evaluation does not address ReDoS-specific scenarios.
Shan *et al.* [112]	Addresses general DoS attacks.
Wei *et al.* [86]	Addresses general DoS attacks.

D Additional Details of the Engineering Review

This appendix provides supplementary information to support the engineering review presented in §4. It includes details on the regex engine selection process and the engine versions used in our measurements.

D.1 Regex Engine Selection

Table 8 summarizes the regex engine implementations reviewed in our ReDoS defenses review (§4). For each programming language considered in our study, we list the selected implementation, notable alternatives, and our rationale. We prioritized (1) engines widely used in production settings (*e.g.*, Node.js—V8, used by 98.8% of websites employing server-side JavaScript [104]); and (2) reference implementations with publicly accessible source code. These are typically cited in official language documentation and serve as the standard for the language (*e.g.*, CPython for Python). These choices ensure both real-world relevance and access to implementation details for assessing ReDoS defenses.

D.2 Versions Used in the Measurements

To ensure the reproducibility of the experiments described in §4.2, Table 9 indicates the specific versions of the programming language runtimes and their regex engines that were evaluated. The *old* versions correspond to earlier releases that predate known or likely ReDoS-relevant mitigations, while the *new* versions refer to more recent releases that incorporate possible ReDoS defenses or performance improvements in their regex engines.

Table 9: Versions used in the ReDoS defense measurements.

Language	Old Version	New Version
JavaScript (*Node.js—V8*)	v15.14.0	v22.2.0
Ruby (*MRI/CRuby*)	3.1.6	3.3.2
C# (*.NET*)	6.0.420	7.0.407
Java (*OpenJDK*)	8u342	23
PHP (*Zend Engine*)	5.6.40	8.3.7
Perl (*perl5*)	5.8.14	5.38.2
Rust (*rustc*)	1.12.1	1.78.0
Go (*gc*)	1.5.4	1.22.4
Python (*CPython*)	3.6.15	3.12.3

Table 8: Regex engine implementations selected for evaluating ReDoS defenses.

PL	Selected Impl. of the PL [Source Code]	Other Alternatives	Reason for Selection
JavaScript	Node.js—V8 [45, 103]	Deno, Bun, SpiderMonkey, JavaScriptCore	Dominates server-side JavaScript usage (98.8% market share) [104]
Ruby	MRI/CRuby [18]	JRuby, Rubinius, mruby, RubyMotion	Reference implementation [17]
C#	.NET [42]	Mono	Reference implementation [24]
Java	OpenJDK [25]	Amazon Corretto, Azul Zulu, Liberica JDK, Eclipse Adoptium...	Most widely used JDK of 2024 with a 20.8% usage rate [91]
PHP	Zend Engine [58]	HHVM, PeachPie, Quercus, Parrot	Standard PHP interpreter is driven by the Zend Engine [59, 60]
Perl	perl5 [99]	N/A (for Perl 5)	Only implementation for Perl 5 [98]
Rust	rustc [49]	mrustc, gccrs	Official and only fully functional compiler for Rust [50]
Go	gc [127]	gccgo, gofrontend, TinyGo, GopherJS, yaegi	Official compiler included in Go releases [128]
Python	CPython [47]	PyPy, Stackless Python, MicroPython, CircuitPython, IronPython, Jython	Reference implementation [46]

PL: Programming Language, Impl.: Implementation (runtime environment, engine, interpreter, or compiler)

E Analyzing Developer Discussions on GitHub ReDoS Issues

In §5, we discussed the idea of "*Developers push back*". We referred to conversations on GitHub and sites of other engineering discourse.

To find discussions about ReDoS vulnerabilities, we used the GitHub REST API to search for issues containing the term `redos` in their title, body, or comments. This approach helped us find both open and closed issues across many repositories, covering a range of cases, including those still active and those already resolved. We started with ~46.4K issues and filtered them to include only those with at least two comments, leaving ~7.4K issues. This helped us focus on discussions with more input. Out of these, ~22.6K issues were not opened by bots. ~4K of those are open, and ~18.6K are closed. We manually reviewed 250 recent issues from each set, along with their comments, to find relevant ReDoS discussions.

Table 10 showcases 10 examples of comments from selected issues, providing URLs for further exploration. These discussions often reveal that developers do not perceive ReDoS vulnerabilities as relevant or critical, leading to delays or reluctance in addressing the issues. While some developers prioritize security, others downplay the severity of these vulnerabilities, resulting in limited attention or resolution efforts. However, we acknowledge this aspect of our investigation is incomplete and bears further study.

Table 10: Selected comments regarding ReDoS and corresponding issue URLs.

Comment	URL
D3 is almost never used to handle user input, so these ReDoS vulnerabilities are generally a huge waste of time.	https://github.com/d3/d3/issues/3939
indeed, but since you'd have to be attacking yourself to trigger the ReDOS in npm, it's not actually a vulnerability here.	https://github.com/npm/cli/issues/7902
None of these seem like they could be exploitable in real world scenarios. Please let us know if you disagree.	https://github.com/facebook/react-native/issues/46996
The few times there was an actual vulnerability, it was reported separately, and we released patches as soon as it was possible. You can always report real vulnerabilities here, but please do this if you understand the difference between a real vulnerability and a false positive. For example, a "Regex DDOS attack" can never be a real vulnerability for a development-time tool. If you're not sure, you're welcome to ask in this thread, but please keep it brief and to the point so that the thread doesn't become unreadable.	https://github.com/facebook/create-react-app/issues/11174
This kind of low-quality finding was assigned a CVE ID without any significant cross-checking.	https://github.com/pytest-dev/py/issues/287
So, a DoS can be caused by a package that's considered for download and installation (which can run arbitrary code), or by the site that serves such packages? I don't see how this can be exploited without the attacker being able to do much more damage; a fix will mainly silence automated checkers.	https://github.com/python/cpython/issues/102202
Most ReDOS vulnerabilities are self-attacks, meaning, not a vulnerability.	https://github.com/sarbbottam/eslint-find-rules/issues/349
Pretty much all complex regexes are vulnerable to ReDos. I've migrated some more important ones to use a token parsing approach instead of regexes.	https://github.com/nodemailer/mailparser/issues/378
When evaluating whether something is a vulnerability, you have to look at the attack vector and the respective cost of upgrading.	https://github.com/facebook/docusaurus/issues/10491
Anyone using any utils with inputs of arbitrary length runs a performance risk. Even built-ins aren't immune.	https://github.com/bestiejs/platform.js/issues/139