# Regulator: Dynamic Analysis to Detect ReDoS

Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna, *University of California, Santa Barbara*

https://www.usenix.org/conference/usenixsecurity22/presentation/mclaughlin

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

# Regulator: Dynamic Analysis to Detect ReDoS

Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, Giovanni Vigna
*University of California, Santa Barbara*
*{robert349, pagani, ncs, chris, vigna}@cs.ucsb.edu*

## Abstract

Regular expressions (regexps) are a convenient way for programmers to express complex string searching logic. Several popular programming languages expose an interface to a regexp matching subsystem, either by language-level primitives or through standard libraries. The implementations behind these matching systems vary greatly in their capabilities and running-time characteristics. In particular, backtracking matchers may exhibit worst-case running-time that is either linear, polynomial, or exponential in the length of the string being searched. Such super-linear worst-case regexps expose applications to Regular Expression Denial-of-Service (ReDoS) when inputs can be controlled by an adversarial attacker.

In this work, we investigate the impact of ReDoS in backtracking engines, a popular type of engine used by most programming languages. We evaluate several existing tools against a dataset of broadly collected regexps, and find that despite extensive theoretical work in this field, none are able to achieve both high precision and high recall. To address this gap in existing work, we develop REGULATOR, a novel dynamic, fuzzer-based analysis system for identifying regexps vulnerable to ReDoS. We implement this system by directly instrumenting a popular backtracking regexp engine, which increases the scope of supported regexp syntax and features over prior work. Finally, we evaluate this system against three common regexp datasets, and demonstrate a **seven-fold increase** in true positives discovered when comparing against existing tools.

## 1 Introduction

Regular expressions (regexps) are a powerful means of expressing complex string manipulation and search operations. Nowadays, regexps are extensively used for a wide variety of purposes, including (but not limited to) data sanitization, intrusion detection systems [15, 46], DNA sequence processing [44, 51], and general-purpose string searching. Previous large-scale community studies reveal that regexps are not only widely known by software developers, but they are also quite popular: 30% to 40% of packages in Python and JavaScript repositories contains at least one regular expression, with an average of two and six regexps per module, respectively [9, 20, 28].

However, despite their popularity and practicality, writing correct regexps represents a challenge for most software developers and maintainers. In the past few years, several studies have shown how regexps are difficult to comprehend and compose, even among experienced users [12, 21, 29, 55]. While several web-based tools and debuggers — purposely created to assist in visualizing and explaining regular expressions — are available [3, 11, 31], users still struggle to compose correct regexps even when using these tools [12]. To overcome these difficulties, users often resort to reusing regexps from online knowledge-bases such as RegExLib [7], or question-answer websites such as Stack Overflow [12, 29]. Unfortunately, this is not always a successful strategy, since regexps are not generally portable across platforms: minor variations in syntax, semantics, and run-time performance can lead to issues including compilation errors and unexpected matching behavior [29].

To make matters worse, matching systems that follow a backtracking construction — such as those found in the standard libraries of Java, Python, Perl, Ruby, and JavaScript — can exhibit worst-case performance that is *super-linear* in the size of the input (subject) string length [29]. Malicious attacks triggering this worst-case behavior are known as Regular Expression Denial-of-Service (ReDoS). Such attacks belong to the class of *algorithmic complexity* attacks, where a malicious input causes a Denial of Service (DoS). ReDoS attacks are particularly impactful in web contexts, and a recent measurement shows that a significant number of websites and web applications are affected by this problem [56]. For instance, in the past few years, multiple incidents related to ReDoS caused extended outages in major online applications and services [35, 57].

Evaluating the worst case run-time performance of a regular expression is not a straightforward task. For example, a seemingly innocuous regular expression (such as `\s+$` or `\d+1\d+2`) can lead to *catastrophic backtracking* when an attacker supplies a carefully crafted string. Moreover, precisely detecting a vulnerable regexp does not only require a deep understanding of how backtrack matching works, but also requires an intimate knowledge of the internals of the specific matching engine on which the regexp is run. This is because regular expressions are not, in practice, evaluated as-is: they are first converted into an intermediary form, undergo several stages of optimization, and are re-emitted in a format useful for lightweight execution. In particular, regexp engines in the family of Spencer-style implementations emit a form of *byte-code*, which is executed by an *interpreter* when the matching procedure is invoked with a particular input string.

Previous research on detecting ReDoS vulnerabilities is based on either static modeling of a representative model [41, 64, 65], or dynamic analysis by fuzzing [39, 47]. Unfortunately, model-based systems are not currently able to capture all extended features of regular expressions, and, more importantly, fail to reflect the low-level details of matching engines. Moreover, regular expression are not a stale technology; new features are continuously proposed and added to popular engines [18]. This represents a problem for static analyzers, because every time a new feature is added, more research is needed to extend these systems. To summarize, no tool makes claims toward completeness or soundness: whether real implementations of regexp engines differ significantly from the idealized models remains an open question.

Another line of research is based on fuzzing, and it focuses on the broader problem of identifying inputs that maximize the algorithm complexity of a given program. Since these systems interact with the real implementation of a regexp engine, they are not affected by the problem of missing features. On the other hand, the fuzzers proposed in prior research are typically general-purpose tools, and their fuzzing logic is unable to effectively leverage the interpreter-based architecture implemented in modern engines. Moreover, the results from fuzzing are difficult to interpret, as users must manually determine how the worst-case performance grows as string length increases.

To address this gap in research, we introduce REGULATOR, a novel dynamic analysis system for detecting ReDoS vulnerabilities. Our system directly instruments a given regexp engine to track its behavior while matching a subject string, and it leverages (and guides) a fuzzer to create inputs with increasingly worse run-time performance. To quickly discover a string that demonstrates a ReDoS attack, REGULATOR's custom fuzzer uses domain-specific mutation strategies. This approach improves on previous research based on automata analysis, because our tool does not rely on any assumptions or approximations of the underlining matching engine, and it supports by design any regexp feature implemented in the matching engine.

To highlight REGULATOR's performance, we evaluate our system on three popular datasets used in previous work (Corpus [20], RegexLib [7] and Snort [8]), and we also analyze more than 40,000 regexps gathered from the 10,000 most popular packages of the npm repository. Our results show a seven-fold increase in the number of vulnerable regular expression found in the wild, compared to previous work. We are currently reporting these vulnerabilities, and 6 CVE numbers have been assigned at the time of writing.
In summary, our paper makes the following contributions:

- We advance the state-of-the-art in ReDoS detection by presenting REGULATOR, a novel dynamic analysis system that leverages fuzzing and domain-specific mutations to detect vulnerable regular expressions.
- We implement REGULATOR against JavaScript's default engine (IRREGEXP), the most used engine in the web-ecosystem where ReDoS represents a real threat to the availability of online services.
- We evaluate REGULATOR on three different datasets used in previous research, and a collection of regexps used in popular JavaScript packages, totaling more than 60,000 regexps.

To foster more research in this field, we have released the source code of REGULATOR along with any other research artifacts at https://github.com/ucsb-seclab/regulator-dynamic.

## 2 Background

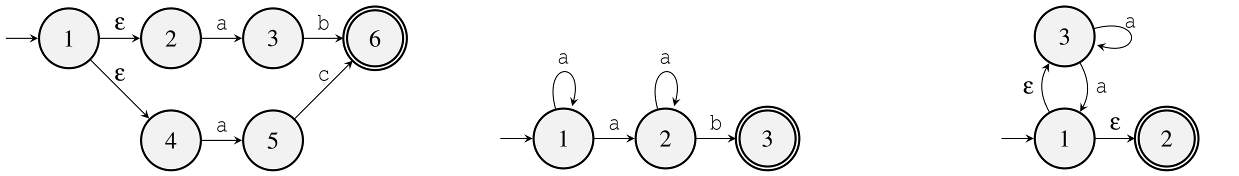
### 2.1 RegExp Syntax and Semantics

Classical regexps can be defined recursively with the following rules:

$$
\begin{aligned}
E &\to c && (\textit{character}) \\
E &\to E_1 E_2 && (\textit{sequence}) \\
E &\to E_1 | E_2 && (\textit{disjunction}) \\
E &\to E_1 * && (\textit{quantifier}) \\
E &\to (E_1) && (\textit{capturing group})
\end{aligned}
$$

Let $\Sigma$ be a finite alphabet of symbols. We say that $\mathcal{L}(E) \subseteq \Sigma^*$ is the *language* of the regexp $E$, which represents the set of strings that are recognized by $E$. Then, we have: $\mathcal{L}(c) = \{c\}$, $\mathcal{L}(E_1E_2) = \{xy | x \in \mathcal{L}(E_1), y \in \mathcal{L}(E_2)\}$, $\mathcal{L}(E_1|E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$, and $\mathcal{L}(E_1*) = \mathcal{L}(E_1)^*$. The *capturing group* operator $(E_1)$ matches the same language as $E_1$, and instructs the matching system to record the exact substring matched by the expression $E_1$.

Most modern regexp engines are extended with additional constructs that go beyond the classical definition of regular expressions. In particular, non-capturing groups, unlike their




(a) NFA for the language {ab, ac}  (b) NFA for the regexp `(a*)a(a*)b`  (c) NFA for the regexp `(a|a*)*`

Figure 1: Automatons accepting various regular languages.

capturing counterparts above, are denoted `(?:E_1)`, and instruct the regexp engine to not record the substring matched by that group. Backreferences, denoted `\n`, assert that whichever substring was matched by the $n^{\text{th}}$ capturing group must appear again at that location. Forward assertions `(?=E_1)` require that the string that immediately follows *must* match `E_1` — but it does not "consume" the string. Likewise, backward assertions `(?<=E_1)` require that the string immediately preceding *must* match $E_1$. Both forward and backward assertions have a negative variant — `(?!E_1)` and `(?<!E_1)`, respectively — which requires that the following string (or preceding string, respectively) *must not* match $E_1$. The word-boundary assertion `\b` requires that the matched position in the string is either the start of the string, the end of the string, or between a "word" and "non-word" symbol. Lastly, the `^` symbol matches the start of a string, and the `$` symbol matches its end.

An important implication of the aforementioned extensions is that $\mathcal{L}(E)$ is not necessarily a *regular language*: for example, the regexp `(a*)b\1` accepts the language $\texttt{a}^n\texttt{b}\texttt{a}^n$, for all $n \in \mathbb{N}$ — which is a *context-free* language [53]. Moreover, modern extensions of regular expressions further broaden the class of accepted languages. For instance, the regexp `(a*)b\1b\1` accepts the language $\texttt{a}^n\texttt{b}\texttt{a}^n\texttt{b}\texttt{a}^n$, which is a member of *context-sensitive* languages. This significantly complicates ReDoS detection: non-regular languages cannot be represented by a finite automaton [53]. Therefore, NFA-based analyses — such as those presented by Weideman [64] and Wüstholz [65] — are unable to process these features. Additionally, string matching in regexps with backreferences is known to be NP-complete [14], and detecting ambiguity of *context-free* languages is known to be undecidable [34]. As a result, static analyses must take great care to avoid (or minimize) the impact of these sub-problems on their results.

For this reason, through the rest of this paper, we use the term "regexp" rather than "regular expression" when referring to the full set of expressions supported by modern engines, extensions included, to avoid any ambiguity of terms.

## 2.2 Backtracking Regexp Engines

In formal language theory, a *regular expression* without language extensions describes a *regular language*, and it is known to always have some finite automaton that recognizes this regular expression's language [53]. Spencer's Algorithm [54] and Thompson's Algorithm [59] are two common approaches to implementing a generalized matching system for regular expressions. Broadly, they correspond to a constrained depth-first search and a breadth-first search through the state-space of a non-deterministic finite automaton (NFA), respectively [29]. An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where $Q$ is the set of states, $\Sigma$ is the set of input symbols (alphabet), $\delta : Q \times \Sigma \cup \{\varepsilon\} \to \mathcal{P}(Q)$ is the transition function, $q_0$ is the initial state, and $F \subseteq Q$ is the set of accepting states [53]. The alphabet is augmented with $\varepsilon$, a symbol indicating a non-deterministic transition consuming no characters.

Spencer-style regular expression matchers are commonly called "backtracking" because the depth-first traversal backtracks upon reaching a no-match condition. To illustrate this behavior, consider the automaton illustrated in Figure 1a, which accepts the language {ab, ac}. When given the string `ac` as input, a depth-first traversal of the automaton may have the following preorder trace: 1, 2, 3, 4, 5, 6. This trace shows how the automaton first attempts to match `ab`. When it reaches node 3, no out-transitions are possible (the `c` in the input does not match the label `b` on the edge from 3 to 6), and the automaton has to backtrack to node 4.

## 2.3 ReDoS Attacks

The core idea behind ReDoS attacks is that certain NFAs exhibit *catastrophic backtracking* under a Spencer-style engine's depth-first traversal.

For example, the NFA illustrated in Figure 1b corresponds to the regular expression `(a*)a(a*)b`. In order to reject strings of the form $\texttt{a}^n\texttt{c}$, the engine must traverse $O(n^2)$ paths. A depth-first traversal at state 1 makes one of two choices upon reading the $i^{\text{th}}$ symbol `a`: either transition to 2, which requires $O(n-i)$ time to reject the match, or self-loop, which presents the same decision recursively for consuming the $(i+1)^{\text{th}}$ symbol `a`. As there are $n$ characters `a`, a simple recurrence relation demonstrates that this requires $O(n^2)$ time to reject. Higher degree polynomial worst-case matching is possible by repeating the pattern [64, 65].

Exponential worst-case complexity is also possible. For example, the NFA illustrated in Figure 1c corresponds to the regular expression `(a|a*)*`. In order to reject strings of the form $\texttt{a}^n\texttt{b}$, the engine must traverse $O(2^n)$ possible paths through the NFA before rejecting the match — as there are $n$ binary choices of state to traverse by consuming a at state ③.

## 2.4 Current ReDoS Detectors

Several state-of-the-art ReDoS static analysis systems base their work upon an NFA representation of the regexp [41, 52, 64, 65]. In their publication, Wüstholz et al. [65] provide a practical algorithm for ReDoS detection in an NFA. Weideman et al. [64] and Liu et al. [41] build upon this with some modification. Below, we provide a brief overview of the patterns within an NFA that indicate potential for ReDoS.

**Polynomial, $O(n^2)$ worst-case.** This detection scheme identifies "loop-branch-loop" structures. This occurs when three paths $P_1, P_2, P_3 \in Q^*$ exist such that all paths accept the same string $w$ and, (1) $P_1$ must start and end at some state $u$, (2) $P_2$ must start and end at some state $v$, and (3) $P_3$ begins at $u$ and ends at $v$. In Figure 1b, this is demonstrated by states ① and ②, where the paths $P_1, P_2, P_3$ all accept the string $w =$ a.

**Exponential, $O(2^n)$ worst-case.** The detection scheme attempts to identify "loop-within-loop" structures within the NFA, where the inner and outer loop begin and end in the same state, and both are satisfied by the same string $w$. In Figure 1c, states ③ and ① exhibit this behavior: the loop ③-①-③ accepts the string a, as does the loop ③-③.

Wüstholz et al. [65] and Liu et al. [41] detect vulnerable structures by depth-first search from the states within $Q$. Weideman et al. [64] leverage the algorithm described by Mohri et al. [10] which performs somewhat faster at the expense of $O(\mathsf{poly}(|Q|))$ memory usage. The output of these systems are an *attack prefix*, *pump string*, and *attack suffix*. The prefix is a string which matches a path from the initial state $q_0$ to the vulnerable component. The pump string is the string $w$ described above, which an attacker repeats to increase the matching time. The suffix string is one which ensures that the sub-match rejects, which forces the engine to backtrack.

Rathnayake et al. [48] take a slightly different approach. Their tool evaluates a *derivation tree* created from the regexp, and search for recursive branching behavior. However, this only identifies $O(2^n)$ vulnerabilities. Finally, Shen et al. [52] dynamically explore an *extended* NFA ($e$-NFA), which supports extended regexp features — however, this method also only identifies $O(2^n)$ vulnerabilities.

# 3 REGULATOR: A Dynamic Analysis System

REGULATOR is a dynamic analysis system with three core components: a feedback-driven generational fuzzer (Section 3.1), a slowdown-pumper (Section 3.2), and a dynamic validator (Section 3.3). Given a particular regexp pattern, REGULATOR is able to detect whether the regexp is vulnerable to ReDoS, and to automatically infer a *pump formula* for generating strings that exhibit super-linear running time.

At a high level, each of the three components performs one task. First of all, the generational fuzzer finds a pathologically slow *witness* string within a fixed budget of $n$ input characters. The witness is then passed to the slowdown-pumper, which infers the pump formula. Finally, the dynamic validator verifies that the pump formula can produce strings that indeed trigger a sufficiently slow execution of the matcher.

## 3.1 Feedback-Driven Generational Fuzzer

*Fuzz testing*, or *fuzzing*, is the process of exploring a program's behavior by repeated execution with varying inputs [43]. Many modern fuzzers are also *feedback-driven*: the fuzzing system uses measurements taken from an input's execution to guide the selection of the next inputs [2, 33, 39, 47, 66]. These systems are typically also *generational*: inputs favored by a heuristic are chosen as parents, and mutated to form the set of inputs in the next generation.

The goal of common bug-finding fuzzers is to discover an input that causes the program to perform an undesirable behavior (i.e., a crash or hang). Program coverage is a natural guiding heuristic for discovering these inputs, and it is used by many fuzzers, including the popular tools AFL [66] and libFuzzer [2].

Prior research shows several effective heuristic techniques for identifying particularly *slow* inputs to a general program. For example, SlowFuzz [47] uses execution path-length as a simple heuristic to guide the generational reproduction. PerfFuzz improves on this strategy by employing a hybrid heuristic based on coverage and path-length [39]. We adopt the latter for our purposes in REGULATOR with a fairly straightforward reapplication of the concepts.

**Instrumentation.** Essential to Spencer's original regexp matching algorithm is an intermediate representation that describes a step-by-step *matching procedure* [54]. As observed by Davis et al. [27], this is primarily an engineering decision above all else, which simplifies the system's construction. This design persists in modern regexp matchers — including Python, Ruby, JavaScript, and Perl — where regular expressions are compiled in the form of a custom *bytecode*, and the matcher operates by executing this bytecode within a lightweight interpreter, using the subject string as input. While in hindsight the intuition of instrumenting the bytecode is straightforward, it represents a fundamental reframing of this problem, and, to the best of our knowledge, we are the first to propose and implement such an approach.

Although the actual implementation of this concept differs across engines, several key design features are shared between different engines. REGULATOR takes advantage of this design by instrumenting the engine's interpreter directly: the matching engine is not treated as a black-box component, but rather instrumented — with a focus on detecting ReDoS. For instance, where normal fuzzers would measure the coverage of the interpreter's code, our tool directly collects the coverage of the bytecode to quickly converge towards a pathologically slow input.

The instrumentation accumulates information within an execution profile ($\pi$) by leveraging the handling procedures of the following two instruction kinds: *character reads* and *branches*. Character reads are instructions that load a single character from the subject string into an interpreter's register. Our system keeps a running record within profile $\pi$ of the last read character index, and it uses this information when handling branch instructions. On the other hand, branch instructions resemble the classical definition of control-flow related constructs: they have two possible successor instructions, and they are guarded by a binary conditional statement. When a branching instruction is executed, REGULATOR increments the total number of times the taken branch has been traversed, and also records in the profile $\pi$ the index of the last character read. Both data points are crucial for REGULATOR's effectiveness, because (1) they measure coverage and (2) they can be used by the mutation procedure to drive the fuzzer towards unexplored paths. Finally, at branching points, we also update a running hash (PATHHASH[$\pi$]) that summarizes the path followed by a subject string into the bytecode program. This hash is used in a later stage of REGULATOR to quickly discard samples that do not exhibit novel behavior.

**Algorithm 1:** Fuzzer Main Loop

**Input:** *seeds S*
1 $C \leftarrow [\ (S_i, \textsc{Run}(S_i)) \mid S_i \in S\ ]$
2 **while** *deadline* **do**
3 $\quad W \leftarrow \textsc{GenChildren}(C)$
4 $\quad$ **for** $w, w' \in W$ **do**
5 $\quad\quad \pi \leftarrow \textsc{Run}(w')$
6 $\quad\quad C \leftarrow \textsc{Merge}(C, w, w', \pi)$
7 **return** $\textsc{MaxCostEntry}(C)$

**Main Fuzzing Loop.** REGULATOR's main fuzzing loop is described in Algorithm 1. The fuzzing process starts by collecting an initial execution profile $\pi$ for each input seed $S_i \in S$ (Line 1), and then enters the main generational loop, which runs until a predetermined deadline (Lines 3 - 6). Once the deadline passes, the main fuzzing loop is terminated and REGULATOR searches the corpus for the maximum cost entry (Line 7), defined as the entry with the highest number of executed bytecode instructions. This entry represents our *witness* string, which is used by the slowdown-pumper in the following phase.

The key components of our fuzzer are the procedures that generate new mutated children and that select the most promising ones. In a nutshell, for each generation within the main loop, our system creates a set $W$ of mutant strings $w'$ paired with the parent string $w$ (Line 3), and collects an execution profile $\pi$ for each mutant in turn (Line 5). The procedure MERGE is then invoked (Line 6), which consults the execution profile $\pi$ to determine whether mutant $w'$ should be discarded or included into the corpus $C$.

The procedures GENCHILDREN and MERGE guide the heuristic exploration of the regexp program's behavior, and are essential to the efficient operation of REGULATOR.

**Algorithm 2:** GENCHILDREN

**Input:** *corpus* $C : \mathcal{P}(\Sigma^* \times \Pi)$
1 selections $\leftarrow [\ ]$
2 **for** $e \in$ BRANCHING EDGES **do**
3 $\quad$ rep $\leftarrow$ *select representative maximizing e in C*
4 $\quad$ *append* rep *to* selections
5 **for** $(w, \pi) \in C$ **do**
6 $\quad$ **if** $w \notin$ selections *and* $\textsc{Staleness}[w] < \mathit{RAND}()$
7 $\quad\quad$ *append w to* selections
8 ret $\leftarrow [\ ]$
9 **for** $w \in$ selections **do**
10 $\quad$ **for** $i \in 0 \ldots \mathit{NumChildren}$ **do**
11 $\quad\quad w' \leftarrow \textsc{Mutate}(w)$
12 $\quad\quad$ *append* $(w, w')$ *to* ret
13 **return** *ret*

**Children Generation.** The routine responsible for generating new mutated children — GENCHILDREN — is described in Algorithm 2. This procedure starts by identifying promising corpus members. To this end, we iterate over all the recorded branching edges and select a *maximizing representative* for each edge $e$ (Lines 2-4). We say that a given corpus member $(w, \pi) \in C$ is *maximizing* for $e$ if there does not exist any distinct corpus member $(w', \pi')$ such that profile $\pi'$ indicates edge $e$ was traversed more than in $\pi$. In the case that multiple distinct corpus entries are maximizing for edge $e$, we select a single representative uniformly at random.

Corpus entries which were not selected as a representative are then considered for inclusion based on the *staleness score* [39] of the input string $w$ (Lines 5-7). *Staleness* is a heuristic that begins low for each entry in the corpus, and increases whenever a mutant offspring of string $w$ fails to produce novel behavior (see Algorithm 3, Lines 4 & 6). This metric is used to discourage the selection of parents that have no record of successful reproduction, while allowing limited exploration of inputs that are not currently maximizing.

We then produce *NumChildren* mutants for each selected parent (Lines 8-13). For a given parent input string $w$ and profile $\pi$, REGULATOR chooses among the following mutations:

- *rotation:* The parent input is rotated either one character left or one character right.
- *crossover:* A co-parent is chosen at random from corpus $C$. Indices $i$ and $j$ are chosen at random such that $0 \leq i <$

$j < n$. The child inherits all characters from $w$, except characters $i$ through $j$, which are inherited from the co-parent.

- *substring replication:* A random substring $\tilde{w}$ is selected from $w$, and replicated elsewhere in the string $w$.
- *arithmetic:* A character index is selected at random and a small value is added (or subtracted) to the character's codepoint value.

Moreover, REGULATOR implements a domain-specific mutation strategy, called *suggestion*, based on the information recorded at branching sites. This mutation selects a random branching edge $e$ (traversed in $\pi$) that originates at a character-based branching instruction. If the string-index of the character under comparison is known, it replaces the character with one that *negates* the branching condition. The rationale behind this mutation is to create mutants that will explore different paths and, therefore, increase coverage. For example, the regexp `abcd(x|\w)*y` is vulnerable to ReDoS, but attack-strings must start with `abcd` in order to reach the vulnerable component. Without the mutation *suggestion*, a purely random-based mutation strategy would take significantly more time to derive the long prefix strings required to explore deeper program states.

Each child input $w'$ is derived by applying a *single* mutation to the parent string $w$. This differs from previous research in this domain, which applies a random number of mutations. We find that our strategy significantly increases the likelihood of discovering novel behavior.

**Algorithm 3:** MERGE

**Input:** *corpus* $C : \mathcal{P}(\Sigma^* \times \Pi)$,
*parent* $w : \Sigma^*$, *child* $w'$, *profile* $\pi'$
1 **if** PATHHASH$(\pi')$ *is unique*
2 **for** $i \in$ BRANCHING EDGES **do**
3 **if** $\pi'$ *is maximal for component i in C*
4 *reset* STALENESS$[w]$
5 **return** $C \cup \{(w', \pi')\}$
6 *increase* STALENESS$[w]$
7 **return** $C$

**Child Merging.** The routine responsible for selectively including mutant children in the corpus $C$ is MERGE, described in Algorithm 3. The algorithm accepts as input the current corpus $C$, a parent string $w$, the mutant child $w'$, and the child's execution profile $\pi'$, collected by the main fuzzing loop (Algorithm 1, Line 5). We begin by quickly exiting from the procedure if PATHHASH$(\pi')$ is not unique within $C$ — indicating that this execution did not contain novel behavior[1] (Line 1). We then include the child *only if* it is maximal for some edge $e$ (Lines 2-5). The combination of these heuristics ensures both that the corpus evolves toward both novel coverage (when the edge $e$ is first traversed) and that the corpus advances toward slower executions within a component (when edge $e$ is traversed a new maximum number of times).

[1]This is uniquely useful in regexp programs, which have relatively small program length compared to general-purpose programs used in typical fuzzers.

## 3.2 Slowdown-Pumper

Prior research into ReDoS makes the observation that attack-strings that trigger worst-case execution time take the form $ab^ic$, for strings $a, b, c$ and $i > 0$ [38, 48, 52, 65]. We call $a$ the *attack prefix*, $b$ the *pump string*, and $c$ the *attack suffix*. Together, we call this a *pump formula*. Attackers may increase the time required to match a string by repeating the pump string $b$ until a sufficient amount of slow-down is achieved.

Fuzzing reveals a slow string within a budget of $n$ characters (the witness string) — but this single data-point is not sufficient for determining whether the worst-case regexp performance is super-linear in $n$. However, we observe that, with very high likelihood, the pump string exists *somewhere* within the witness string. For example, the regexp `abc(123|\d)*x` is vulnerable to the prefix `abc`, pump string `123`, and suffix `a`. After fuzzing for only 2 seconds with $n = 13$, REGULATOR discovers the witness string `abc123123123a` — which clearly contains the pump string. In what follows, we take inspiration from Shen et al. [52] and use a simple yet effective strategy to identify the pump string by a heuristic scan over the witness.

**Algorithm 4:** GENPUMPFORMULA

**Input:** *witness* $w : \Sigma^*$
1 candidates $\leftarrow$ []
2 slowest_per_char $\leftarrow -\infty$
3 **for** len $\in 1 \dots n$ **do**
4 **for** pos $\in 0 \dots (n -$ len$)$ **do**
5 prefix $\leftarrow w[0 :$ pos$]$
6 pump $\leftarrow w[$pos : pos $+$ len$]$
7 suffix $\leftarrow w[$pos $+$ len $: n]$
8 attack $\leftarrow$ prefix $+$ pump$^{\text{NPUMPS}}$ $+$ suffix
9 $t \leftarrow$ TIME(attack)
10 **if** $t$ / len> slowest_per_char
11 slowest_per_char $\leftarrow t/$len
12 $\phi \leftarrow$ MODELFIT$(w,$ pos, len$)$
13 *append* (pos, len, $\phi$) *to* candidates
14 **return** (pos, len, $\phi$) *for steepest* $\phi$ *in* candidates

The routine is described in Algorithm 4. We begin by iterating over every substring of the witness $w$ in order of increasing length (Lines 3-4). We then extract a *prefix* (all characters prior to the substring), *pump string* (the substring considered), and *suffix* (all characters after the substring), and construct a candidate *attack* string by repeating the pump string a large number of times (Lines 5-8). We then run the regexp matching procedure against the attack string, and measure the number of bytecode instructions executed during matching (Line 9). We use the heuristic "instructions added per pump character" to quickly discard pump-string candidates that do not cause a larger slow-down than any prior candidate (Line 10). Finally, we perform a model fitting procedure to estimate the growth function of instructions executed as the string is pumped (Line 12). The model fitting procedure works by regression. We take measurements of the matching time for attack strings formed by repeating the pump string from 10 to 256 times, increasing by 13 each iteration, for a total of 20 measurements. We then fit the data against three models: (1) linear: $\hat{t} = \alpha n + \beta$, (2) power: $\hat{t} = \alpha n^{\beta}$, and (3) exponential: $\hat{t} = \alpha e^{\beta n}$. We ignore the power regression when $\beta \approx 1$, as this is likely better explained as a linear growth model. We select $\phi$ to be the model with the highest $r^2$ coefficient.

The procedure completes by selecting the pump substring with the steepest growth model (Line 14). Preference is given first to exponential models, and second to power models. In both cases, ties are broken by the highest value $\beta$.

### 3.3 Dynamic Validator

The final stage of REGULATOR is a dynamic validator. We adopt a simple metric used in prior research: we say that a regexp is vulnerable to ReDoS if an attack string of less than 1 million characters can cause 10 seconds or more time spent in the matching system [28, 29]. We verify that an attack string derived from the pump formula meets this criteria. Furthermore, if enabled by configuration, we use a binary-search between the length of the witness string length $n$ and the length of the attack string to find the smallest string that results in at least 10 seconds of matching time.

## 4 Implementation

In this work, we stray from the tradition of polyglot ReDoS detection and instead exclusively focus on regexps as implemented by a specific matching engine. In particular, we focus on IRREGEXP, a popular drop-in library for compiling and evaluating regular expressions. This matching engine is currently used by Mozilla's Firefox web browser [37] and the V8 JavaScript runtime [26], which sits behind both Google's Chrome web browser and the NodeJS language runtime.

**NodeJS and ReDoS.** NodeJS is a server-side JavaScript runtime with a single-threaded, event-driven design. The runtime processes work by selecting events one at a time from an *event queue*. Once an event is selected, its event handler function is invoked with exclusive, non-preemptible execution. Several researchers have observed that these single-threaded, event-driven languages suffer from a class of Denial-of-Service attacks known as Event Handler Poisoning (EHP) [30, 45]. EHP occurs when an attacker is able to cause significant delay in the event handler's execution thread, which prevents any further events from being processed until the victim thread yields. In this scenario, ReDoS is a particularly problematic type of event handler poisoning, as there does not exist any method of preempting a thread from the regexp matching subsystem — effectively halting all progress until the match procedure completes, and immediately reducing server throughput to zero. To address this serious concern, we will focus on the implementation of REGULATOR for the IRREGEXP matching engine. However, our methodology remains general and applicable to other backtracking engines with sufficient similarity.

| Instruction | Args. | Description |
|---|---|---|
| `PushBt` | `addr` | Push code address `addr` onto the stack |
| `PopBt` | - | Pop an address from the stack and jump to that address |
| `PushCp` | - | Push the value of `cp` onto the stack |
| `PopCp` | - | Pop a value from the stack and store it in `cp` |
| `Fail` | - | Halt and reject the string |
| `Succeed` | - | Halt and accept the string |
| `GoTo` | `addr` | Jump to address `addr` |
| `LoadCurrentChar` | `i`, `addr` | Set register `lc` to the character at `cp + i`. Jump to `addr` if that index is out of range. |
| `CheckChar` | `c`, `addr` | Jump to `addr` if `c = lc` |
| `AdvanceCP` | `i` | Set register `cp` to `cp + i` |
| `AdvanceRegister` | `ri`, `i` | Set register `ri` to `ri + i` |
| `SetRegister` | `ri`, `v` | Set register `ri` to `v` |
| `SkipUntilChar` | `c`, `i`, `j`, `addr` | Set register `cp` to `cp + i` until the character at `cp + j` equals `c`. Go to `addr` if `cp + i` is out of range. |

Table 1: Selected instructions of the IRREGEXP interpreter

**Irregexp Internals.** IRREGEXP is a Spencer-style backtracking regexp matching engine [26, 29] that works in two phases. In the first one, the user invokes the IRREGEXP compiler with a regexp as input (the *pattern*). The compiler emits a *regexp program* — i.e., a code simulating a depth-first exploration of the regexp state space. These programs can be either bytecode, which is executed by the IRREGEXP interpreter, or platform-native code. The semantics of these programs are completely equivalent and, for simplicity, we chose to analyze the *bytecode* regexp programs. In the second phase, the user passes an input (*subject*) character string, and IRREGEXP executes the *regexp program* with the *subject* as an input. If the regexp program halts at a `Succeed` instruction, then the system reports a match; otherwise, the regexp program halts at a `Fail` instruction, and the system reports a non-match. All regexp programs are halting, by design of the IRREGEXP compiler.

The IRREGEXP interpreter's memory model is an expandable stack of 32-bit integers, a finite set {r0,...,rn} of general-purpose 32-bit registers (where n is determined at compilation), a 32-bit character index register cp, and a loaded character register lc. The interpreter does not allow arbitrary stack reads — only the topmost 32-bit integer may be examined. [2]

The IRREGEXP interpreter supports 59 instructions, some of which are summarized in Table 1. In general, the instructions fall into one of four categories: stack manipulation, conditional branching, register manipulation, and fused loops. An example of the latter category is SkipUntilChar, which advances the current position in the string (cp) until some character is seen at a given offset. This instruction, in combination with others, is used to implement the Boyer-Moore fast string search algorithm [17].

We provide a simplified disassembly of the regexp program for ^ab*$ in Listing 1. Instructions 0x1c-0x24 set up a stack frame: PushBt places a return address on the stack, and PushCp stores the character index register cp on the stack, so it can be restored after returning from the procedure call. Instructions 0x3c-0x40 are, likewise, similar to a function return. When executed, after consuming the character a, character b is repeatedly consumed by the recursive procedure at instructions 0x2c-0x54. If any character other than b is read, then the program begins a cascading return (PopBt) until it reaches Fail and exits. Otherwise, the program will eventually reach the end of the string at instruction 0x2c, which branches to Succeed and exits.

Listing 1: Disassembly of ^ab*$

```
 0x0  PushBt           addr: 0x60
 0x8  LoadCurrentChar  i: 0        addr: 0x18
0x10  CheckChar        c: 'a'      addr: 0x1c
0x18  PopBt
0x1c  PushCp
0x20  AdvanceCp        i: 1
0x24  PushBt           addr: 0x18
0x2c  LoadCurrentChar  i: 0        addr: 0x5c
0x34  CheckChar        c: 'b'      addr: 0x44
0x3c  PopCp
0x40  PopBt
0x44  PushCp
0x48  AdvanceCp        i: 1
0x4c  PushBt           addr: 0x3c
0x54  GoTo             addr: 0x2c
0x5c  Succeed
0x60  Fail
```

**Instrumentation.** A important design feature of REGULATOR is that it requires minimal instrumentation of the target engine. Similar to other regexp engines, the central component of IRREGEXP's interpreter is a switch-case dispatcher that invokes instruction handlers. For each instruction, we added a call to the function responsible for updating the execution profile $\pi$. To instrument branching instructions, we also retrieve the target address, either from an argument or from the top of stack. Character reads are handled similarly, since they also accept an offset from the current character index cp as an argument.

[2]This makes the IRREGEXP interpreter *not* Turing-powerful.

| Name | Source | Ref. | Size | Description |
|---|---|---|---|---|
| | Corpus | [19] | 13,597 | OSS Python projects [20] |
| *base* | RegExLib | [49] | 2,990 | Online regexps website |
| | Snort | [50] | 10,037 | Rules used in the Snort IDS |
| *npm* | Node Registry | [32] | 42,743 | Popular JavaScript packages |

Table 2: Datasets used in our evaluation.

In total REGULATOR's fuzzer required only 3,000 lines of C++ code, 200 of which are for instrumentation, and the remainder implement the fuzzer's control procedures.

## 5 Evaluation

In this section, we evaluate how REGULATOR performs on a diverse set of regexps. In particular, we set out to answer the following research questions:

- **RQ1**: Does the overall architecture of REGULATOR help to quickly explore the state-space of regular expressions (when they execute as programs on top of the matching engine)?
- **RQ2**: Is REGULATOR effective in finding ReDoS vulnerabilities, and does it do so better than previous work?
- **RQ3**: Does REGULATOR discover previously unknown ReDoS vulnerabilities in real-world packages?

**Dataset.** As summarized in Table 2, we select two different datasets to perform our evaluation. The first dataset contains regexps from three different sources, which we chose because of their extensive use in previous ReDoS research (*base dataset*). The second dataset contains regular expressions extracted from real-world JavaScript packages (*npm dataset*). To collect the *npm dataset*, we downloaded the top 10,000 most popular packages from the NPM package registry, as measured by monthly downloads. JavaScript allows users to express regular expressions in two forms. The first one is by a language literal, i.e., /pattern/flags. To find such regular expressions, we use a traversal of each source file's abstract syntax tree (AST) and extract both the pattern and, if present, any modifier flags (such as i to enable case-insensitive match). This method produced 40,877 distinct regexps. Moreover, users may also invoke the constructor RegExp(pattern, flags) to create a regexp at runtime. To handle these cases, we use the code analysis engine CodeQL to perform a data-flow analysis on the package's source code [4]. We record all known strings that are used to construct a regular expression in this manner. This analysis produced an additional 2,019 distinct regexps.

**Test Environment.** All experiments were performed on an Intel Xeon Gold 6252 CPU, with 377 GB total RAM, running Ubuntu 20.04. Each tool was assigned a unique CPU core with `taskset` to avoid interference in results.

### 5.1 Does REGULATOR quickly explore program state-space?

Berglund et al. [14] recently published a proof showing that string matching in regexps with backreferences is NP-complete. Furthermore, deciding whether *any* string is accepted by a regexp with backreferences is also in NP [14]. For this reason, we will use *program coverage* as a tractable vehicle to evaluate exploration of the program state-space.

We run REGULATOR's fuzzer for five minutes on all 69,367 regular expressions. During these executions, we regularly sample the total coverage of the regexp program, as measured by percentage of instructions executed at least once during the fuzzing session. In order to explore how the size of the regexp program impacts the attained coverage, we categorize each regexp into either *small*, *medium*, or *large*, depending on the number of instructions in the regexp program. We create each category to contain exactly ⅓ of the dataset. The category instructions bounds are provided in the following table:

| Small | Medium | Large |
|---|---|---|
| 0 - 51 | 52 - 101 | 102 - 66,676 |

Quite interestingly, two-thirds of the regexps used in the wild compile to programs with less than 101 instructions, and 95% of them contain less than 435 instructions. The median coverage of our fuzzer, over time, is displayed in Figure 2. This experiment shows that, even among large-sized regexp programs, our fuzzer rapidly achieves over 80% coverage within 10 seconds. Moreover, all categories reach more than 90% median coverage within the five minute time bound. While investigating the results of our fuzzer, we quickly realized that some regexps produce programs that include *unreachable* code. For instance, every regexp program includes by default some code to support "sticky" mode (flag 's') that, when enabled, allows the user to *resume* the regexp interpreter after it exits with a successful match. The code to support this feature is present even when sticky mode is disabled, and it is entirely unreachable. Another example is regexps that contain a disjunction, where the first element matches all strings which are matched by the second (for example, `\d|1`). In this case, the instructions to match the second element will be emitted, but never executed. In other words, the results presented in this section represents a lower bound on REGULATOR's performance. To summarize, the previous results clearly show that we can answer **RQ1** in the affirmative: REGULATOR's heuristics are effective in exploring the program state-space.

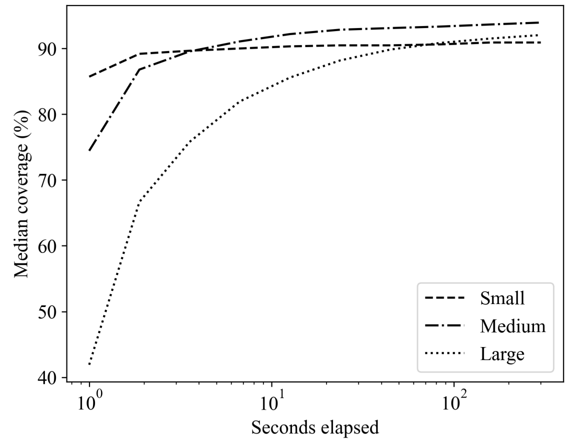



Figure 2: Median coverage of REGULATOR-fuzzer over time, separated by size of regexp program as determined by instruction count.

### 5.2 Comparison with NFA-based approaches

To compare REGULATOR with NFA-based approaches, we selected five popular tools that detect ReDoS using NFA-based analysis: RXXR2 [48], Rexploiter [65], NFAA [64], ReScue [52] and Revealer [41].

In order to make a sensible comparison, we must first introduce the concept of *full-match* and *partial-match* semantics. A regexp engine performs full-match semantics if the entire string must satisfy the regexp pattern in order to accept the input string. On the other hand, matchers using partial-match semantics will accept a string if any *substring* of the input satisfies the regexp pattern. For example, the regular expression `c.r` would match `car` but not `carpet` using full-match semantics, but partial-match semantics would match in both cases.

As noted by Davis et al. [29], all the systems we use in our evaluation assume full-match semantics. However, this is not always appropriate: for example, Python's regexp library allows the user to select either full-match or partial-match semantics (i.e., by using the methods `match` or `search`), while JavaScript's matcher offers only partial-match semantics. This difference has profound implications in terms of backtracking complexity, since the matching semantics *does* impact running-time performance. For instance, the regexp `a*b` exhibits $O(n)$ worst-case running time when using full-match semantics, but $O(n^2)$ worst-case running time when using partial-match semantics.

In order to level the field and to run a meaningful comparison across tools, we conform REGULATOR with the majority

| Tool | Corpus | | | | RegExLib | | | | Snort | | | | Total | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Sup. | TP | FP | FN | Sup. | TP | FP | FN | Sup. | TP | FP | FN | Sup. | TP | FP | FN |
| RXXR2 | 11,696 | 30 | 26 | 2,154 | 2,301 | 100 | 27 | 451 | 7,102 | 10 | 5 | 2,200 | 21,099 | 140 | 58 | 4,805 |
| Rexploiter | 10,536 | 293 | 973 | 1,891 | 1,764 | 98 | 287 | 453 | 5,795 | 53 | 1,035 | 2,157 | 18,095 | 444 | 2,295 | 4,501 |
| NFAA | 11,256 | 738 | 952 | 1,446 | 1,977 | 279 | 70 | 272 | 6,169 | 831 | 154 | 1,379 | 19,402 | 1,848 | 1,176 | 3,097 |
| ReScue | 12,441 | 26 | 4 | 2,158 | 2,780 | 115 | 14 | 436 | 7,765 | 8 | 0 | 2,202 | 22,986 | 149 | 18 | 4,796 |
| Revealer | 13,206 | 428 | 30 | 1,756 | 2,946 | 245 | 17 | 306 | 10,035 | 232 | 4 | 1,978 | 26,187 | 905 | 51 | 4,040 |
| **REGULATOR** | **13,595** | **2,156** | **0** | **28** | **2,973** | **519** | **0** | **32** | **10,037** | **2,172** | **0** | **39** | **26,605** | **4,847** | **0** | **99** |

Table 3: ReDoS detection results for the *base* dataset.

of existing tools. That is, we evaluate the *base* dataset using full-match semantics. Since JavaScript only offers a partial-match method, we anchor each regular expression with the caret `^` and dollar `$` signs (e.g., `^(\d+1\d+2)$`)[3]. We believe this experiment design provides a useful comparison between the tools.

### 5.2.1 Base Dataset

We run RXXR2, Rexploiter, NFAA, ReScue, and Revealer on each regexp in the base dataset. We allow each tool up to 10 minutes and 16 GB of memory to analyze each regexp — both limits are far more generous than what has been used in previous research [28, 29] — and we record the pump formula reported by each tool. REGULATOR is also allowed the same amount of time to run: we divide the time evenly among fuzzing and slowdown-pumping, and we set aside 10s for the dynamic validation phase. We configure the fuzzer to find witness strings at a fixed length of 200 characters, starting from a seed containing only the character `a`.

Each pump formula reported by the aforementioned tools is then validated to determine whether it is a *true positive* or *false positive* result. Since there is no general consensus about what constitutes a true or false positive, we decided to use the heuristic employed by most prior work, which states that a result is a true positive if the pump formula produces 10 seconds of delay within a budget of 1 million characters [28, 29]. If we fail to produce 10 seconds of delay, we label the result as a false positive. Identifying *false negatives* involves slightly more nuance. Our datasets are not labeled, so we must resort to the following strategy: Given a regexp, we say that any tool reports a false negative whenever it does not report a ReDoS vulnerability, but at least one other tool reported a true positive result for that regexp. In other words, since we can unequivocally demonstrate that a regexp is actually vulnerable, we build the set of true positives by combining the vulnerabilities reported by at least one tool.

Table 3 summarizes the results for each dataset contained in the *base* dataset. We can see that REGULATOR finds at least two to three times more true positives (vulnerable regular expressions) when compared to the state-of-the-art ReDoS detection tools. Moreover, REGULATOR eliminates false positives by design, since pump formulas are dynamically tested before any report is raised. REGULATOR's dynamic validator only rejected 114 (2.4%) pump formulas. Finally, our tool demonstrates a false negative rate two orders of magnitude lower than all other tools.

During this experiment we observed 3,059 timeouts from ReScue, and 350 timeouts from NFAA. ReScue's original publication used a 10 minute timeout [52], and NFAA used a 10 second timeout [64] — so we believe that our time limit of 10 minutes is a fair comparison. For both tools, ⅓ of the timeouts occurred while analyzing known-vulnerable regexps.

**Selected Results.** REGULATOR's remarkable effectiveness is illustrated by the following selected examples, which lists several ReDoS detections that are unique to our tool:

```
%module(\s*\(.*\))?\s+("?)(.+)\2
(?:\b\w*(\w\w?)\1{2,}\w*\b)
dir\s*=\s*[\x22\x27]?a((?!^--).)*?\x2e\x2e[\x2f\x5c]
```

In the first example, we demonstrate an effective handling of backreferences. ReScue does handle backreferences, but failed to find this vulnerability. Revealer and RXXR2 only handle backreferences by approximation (by ignoring them), and likewise did not identify this vulnerability. On the other hand, the second example demonstrates a precise handling of *quantified* backreferences. The structure `\1{2,}` is only supported by ReScue, but their method cannot find this $O(n^2)$ super-linear regexp. This highlights the power of REGULATOR's approach: no additional work beyond initial instrumentation was required to support this syntax and semantics. Finally, the last example demonstrates awareness of *flags* passed to the regexp matching system — in this case, the flags used were 'm', 'i', and 's'. This example clearly shows how flags should not be dismissed, since this pattern is *only* vulnerable when the 'm' flag is set, indicating multi-line mode. Among previous research, the only other tool that recognizes regexps flags is Revealer, which did not identify this vulnerability.

[3]We first recursively remove all anchors from within disjunctions. For example, `^a|(b$|c)` becomes `^(a|(b|c))$`.

| **Tool** | **NPM** | | | |
|---|---|---|---|---|
| | Sup. | TP | FP | FN |
| RXXR2 | 35,842 | 120 | 88 | 5,969 |
| Rexploiter | 30,317 | 77 | 1,135 | 6,040 |
| NFAA | 32,158 | 813 | 1,411 | 5,280 |
| ReScue | 39,258 | 143 | 4 | 5,946 |
| Revealer | 38,379 | 410 | 5 | 5,676 |
| **REGULATOR** | **41,342** | **5,954** | **0** | **132** |

Table 4: NPM ReDoS Detection Results

Once again this result is immediate from our approach: no additional work was necessary to support these flags.

### 5.2.2 NPM Dataset

We conduct a similar comparison with the NPM dataset, by running each tool with the same time and memory limits used in the previous experiment. This time, however, we *do not* anchor the regexp — neither while testing the regexp nor while evaluating the pump formula. This choice was made both to demonstrate effectiveness with partial-match semantics, but also to most closely mirror how the regexps are used in their original setting (recall that NPM packages are written in JavaScript, which uses partial matching semantics). As in the last experiment, we verify that each pump formula is able to achieve 10 seconds of delay within 1 million characters.

The results are displayed in Table 4. In this experiment, we demonstrate a *seven-fold* increase in true positive detections over the next-best analysis tool. REGULATOR was able to identify *several thousand* additional true positives, and was able to run against every syntactically valid regexp. REGULATOR's dynamic validator only rejected 16 (0.3%) of pump formulas.

The surprising performance of REGULATOR is indirectly confirmed by the findings reported by Davis et al. [28]. In their paper, the authors tested RXXR2, NFAA, and Rexploiter, against a dataset of 349,852 regular expressions, which was extracted from more than 500,000 NPM packages. They were able to classify and verify 3,589 regexps as having super-linear worst-case run-time. On the other hand, our tool was able to identify 5,954 super-linear regexps, even though we analyzed a considerably smaller amount (10,000) of packages, and a significantly smaller number of regexps.

**Selected Results.** The following list of regexps show some examples of novel detections by REGULATOR on the NPM dataset:

```
<<-?(\w+\b)[\s\S]*?^[ \t]*\1
.*rst$|
^['"]+|['"]+$
```

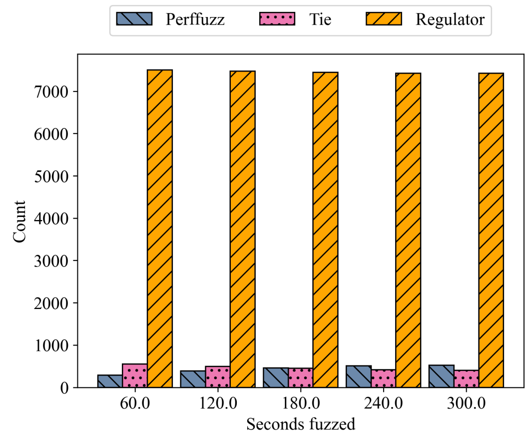



Figure 3: Count of regexps for which each fuzzer has the maximum path input over time, known vulnerable regexps only. Ties are ± 100 instructions.

The first regexp, which was combined with the multi-line flag ('m'), demonstrates REGULATOR's intrinsic deep knowledge of the underlying matching system: when this flag is set, the start-of-string character ^ changes meaning, and instead matches the empty string preceded by a newline character, `\n`. The second example highlights instead the importance of analyzing the correct semantics — this regexp was considered safe by all other tools, but, in fact, it exhibits super-linear behavior when run in partial-match semantics. Finally, the last example takes the aforementioned features one step further, and demonstrates that the start (`^`) and end (`$`) anchors can appear combined within a disjunction (`|`). Once again, REGULATOR was the only tool to successfully handle such complex matching semantics.

## 5.3 Comparison with PerfFuzz

In this section, we examine whether REGULATOR produces better results when compared with existing slow-down fuzzers. For this comparison, we select PerfFuzz, a state-of-the-art general-purpose program fuzzer for finding pathologically slow inputs [39]. We compile and instrument a standalone executable version of IRREGEXP using the PerfFuzz tooling. To run a fair and meaningful comparison, we reproduce the same setting used by our fuzzer: the regexp program is compiled outside of the fuzzing loop, and the executable loads the regexp program before running the matching procedure. Both fuzzers are given five minutes for each regexp, and both are configured to produce inputs no larger than 200 characters. We record the subject strings covering the highest number of instructions as they are discovered by each fuzzer. We run this evaluation against the *base* dataset.

Figure 3 shows, as the time progresses, whether REGULATOR or PerfFuzz found the input that causes the highest number of instructions to execute. From this figure, we can

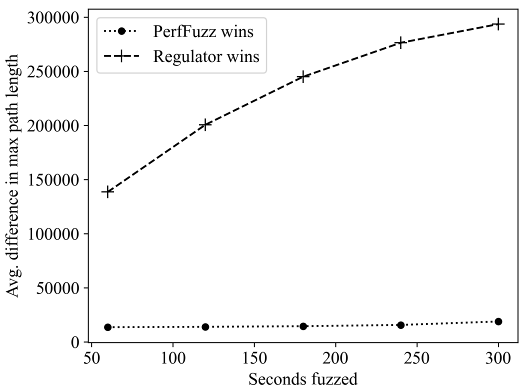



Figure 4: Average difference in maximum path-length discovered over time, separated by fuzzer with the slowest input, known vulnerable regexps only.

clearly see that REGULATOR finds the slowest input string for nearly all regexps, at most time-points. More interestingly, Figure 4 shows the average *difference* in program path-length, on a per-regexp basis, that each fuzzer discovers. This result highlights how the longest paths found by REGULATOR are *significantly* slower than the paths found by PerfFuzz — whereas, when PerfFuzz finds the longer path, it is only longer by a small margin.

This success is because REGULATOR's interpreter-level coverage feedback and domain-specific mutations create an effective regexp program exploration. For example, the *suggestion* mutation is informed by previous executions of the regexp, which helps uncover components that can only be executed by correctly guessing a series of characters.

We then take the slowest input discovered by PerfFuzz and feed it to our slowdown-pumper subsystem to derive a pump formula, with a 5 minute time limit per regexp. This resulted in 4,224 pump formulas with verified 10 second delay, compared with 8,713 from REGULATOR[4]. PerfFuzz only discovered 42 vulnerable regexps which REGULATOR failed to identify, which we discuss below.

The success in generating pump-formulas demonstrates that REGULATOR's witness string is not only much slower, but is also much more likely to exercise ReDoS-vulnerable behavior, which is automatically identified and generalized to a formula by the slowdown-pumper.

We now conclusively answer **RQ2** in the affirmative: REGULATOR is significantly more effective at finding ReDoS vulnerabilities in comparison with previous analysis systems, including a general-purpose slow-down fuzzer.

[4]This number is different from the result reported in Table 3 because in this experiment we evaluate using partial-match semantics.

## 5.4 Real-World Vulnerabilities

We are currently working with NPM package maintainers to address vulnerabilities discovered in this work. So far, 10 vulnerabilities have been acknowledged and fixed, and 6 CVE numbers were assigned. Packages for which vulnerabilities have been acknowledged range from 1 million to 100 million monthly downloads.

An interesting vulnerability reported by our tool, is CVE-2021-23425 [1], which is part of a popular string processing library, and has $O(2^n)$ worst-case time complexity. The vulnerable regular expression is the following: `^(?:\r\n|\n|\r)+|(?:\r\n|\n|\r)+$`, and is used to match newlines, either at the beginning or end of a string. Subject strings that attack this regexp begin with the prefix `a`, which ensures that the first component of the disjunction rejects, and the second component is evaluated. The pump string in this case is `\r\n`: which can match either as `(\r\n)` or `(\r)(\n)`. Finally, the suffix string is the character `a`, which ensures that the matcher must attempt all $O(2^n)$ possible combinations of the pump string before the string is finally rejected. We observe that repeating the pump-string just 25 times causes 14 seconds of delay, and this time *doubles* with each repetition. Repeating the pump-string a mere 80 times would cause delay of approximately 16 billion years, permanently reducing program throughput to zero. At the time of writing, this package receives approximately 10 million downloads per month, and is transitively included by thousands of other packages in the NPM ecosystem.

Figure 5 plots a cumulative distribution function (CDF) of the minimum known string lengths that cause 10 seconds of delay for all vulnerable regexps in our dataset. The median length is 65 KB, which indicates that the budget of 1 million characters was extremely generous, as most vulnerable regexps cause significant delay with under 10% of that limit.

We now answer **RQ3** in the affirmative: REGULATOR is effective at discovering previously unknown vulnerabilities in real-world packages.

## 5.5 Limitations

**Invalid pump formulas.** In these experiments, the slowdown-pumper generated some pump-formulas that did not pass dynamic validation. Upon manual inspection, we find that these primarily arise from two root causes. First, some regexps exhibit super-linear behavior at lengths tested by the pumper, but become benign at even larger lengths. For example, the complexity of `ab*\w{1024}c` is $O(n^2)$ when $n = 1024$, but $O(n)$ $n > 1024$. Second, some regexps exhibit super-linear behaviour, but the running-time grows too slowly to exploit — this primarily occurs when the *pump sting* is very long.

**False Negatives.** In the prior two experiments, REGULATOR failed to identify 141 vulnerable regexps. In the comparison

| | REGULATOR | Revealer | ReScue | RXXR2 | Rexploiter | NFAA |
|---|---|---|---|---|---|---|
| Backreferences `\i` | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| Lookarounds `(?=)` | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Non-capturing groups `(?:)` | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Named groups `(?<Name>)` | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Unicode beyond `0xFFFF` | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Word Boundary `\b`, `\B` | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Greedy and lazy quant. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Flags | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |

Table 5: Semantics supported by ReDoS detection systems, based on the results presented by Liu [41] and Shen [52]. REGULATOR supports all semantics of IRREGEXP.

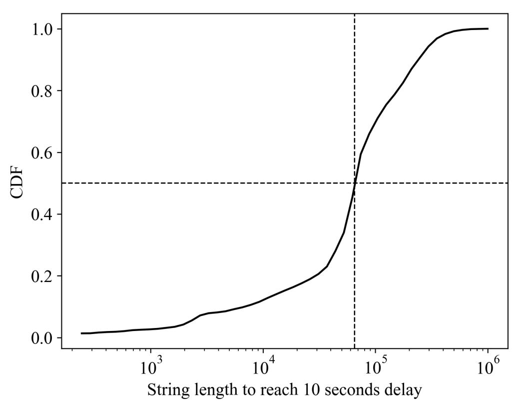



Figure 5: CDF of attack string length required for 10 seconds of delay. Vertical and horizontal lines are drawn to show the median (65 KB)

with PerfFuzz, 23 of these were caused by REGULATOR's fuzz witness being *much slower* than the one found by PerfFuzz. This caused the slowdown-pumper to time out, as evaluating each candidate *pump string* took more time. In other words, despite our tool found a valid candidate, we are unable to synthetize a pump formula. In the comparison to NFA-based tools, 52 false-negatives were also caused by a timeout in the slowdown-pumper. To verify this cause, we introduced an exponential backoff to the fuzzing phase: if the slowest input exceeds 500,000 instructions in path-length, then we restart the fuzzer, configured to find a witness one-half as long. The goal of this is to produce witness strings which run faster, but still demonstrate ReDoS vulnerability. Upon a rerun with this new technique, REGULATOR was able to identify a valid pump formula for all these 75 regexps.

The remaining false-negatives are related to two main causes. First, some regexps contain two slow components, both of which are super-linear at 200 characters, but only one is super-linear at larger lengths. If the witness string only exercises the slow component that is not vulnerable at larger lengths, then no vulnerability will be identified. Second, some regexps' running-time growth functions are not smooth, and are difficult to classify by measurement and regression. If the pumper does not have a strong model-fit for a super-linear growth function, then no vulnerability will be identified.

## 6 Related Work

**ReDoS Static Detection.** The first tool based on static analysis of regexp to detect ReDoS was *safe-regex*, released by Halliday [36] in 2013. This tool calculates the star height of a regular expression, which is the maximum depth of nested star quantifiers, and reports a vulnerability when this number is greater than 1. While this tool is extremely fast, the star depth condition is only necessary but not sufficient, and therefore *safe-regex* has low recall and precision [28].

In the next several years, the academic community proposed several more sophisticated approaches: RXXR2 [48], Weideman's NFAA [64], Wustholz's Rexploiter [65], and most recently Liu's Revealer [41]. These analyses involve significant formal theoretical work, but fail to capture the entire set of features available in modern regexp engines. The results presented in this paper highlight the limitations of static approaches, and demonstrate that dynamic analysis is an effective alternative.

**ReDoS Dynamic Detection.** The first attempt to dynamically detect ReDoS attacks was presented by Sullivan [58], with the tool *SDL Regex Fuzzer*. This tool automatically generates subject strings and times how long the matching engine takes to process it. SDL Regex fuzzer derives its inputs first by creating a set of subject strings that *match* the regexp under test, and then applying a single mutation strategy — appending a random character to the end of each string. ReScue [52] operates on a similar theory, but dynamically explores an *extended* NFA based on Java's regexp engine, with smarter mutation strategies. While this technique may be enough to find exponentially vulnerable regexps, it does not suffice to identify polynomial backtracking, where a pump string must be repeated hundreds of times before the matching engine shows a considerable slowdown.

**ReDoS Prevention.** Different approaches have been proposed to prevent ReDoS attacks. The first line of research is based on automatically transforming vulnerable regexps into safe ones [23,24]. For instance, van der Merwe et. al. [60] proposed a series of techniques to reduce or remove the root cause of ReDoS attacks — i.e., ambiguity during matching. Cody-Kenny et. al. [25] proposed to improve the performance of regular expression using a genetic programming algorithm. Li et al. [40] presented *FlashRegex*, a technique that is able to deduce safe regexps by either synthesizing or repairing existing ones, starting from a set of matching subject strings. Another line of research focuses on modifying the matching engine to avoid super-linear behavior. Davis et al. [27] proposes *selective memoization* to obtain linear time matching, albeit by increasing the space complexity. Rust's regexp engine [6] and Google's stand-alone engine RE2 [5] both offer guaranteed linear matching time. However, neither of these support extended syntax features. Finally, Davis et al. [28] identified three super-linear regex anti-patterns that the authors suggest to avoid, to reduce the likelihood of writing a vulnerable regexp.

**Empirical Studies.** The interactions between regular expression, developers, and the software development process have been extensively studied in previous research [13,22,42,63]. Chapman et al. [20] explored the context and the features of regular expressions used in Python projects. Davis et al. [29] report that regular expression re-usage is prevalent among developers, and how this practice can lead to semantic and translation problems. Wang et al. [61,62] studied how regular expression bugs are fixed, based on pull requests of popular open-source projects.

**Algorithmic Complexity.** The most promising technique to find inputs that causes slowdowns is PerfFuzz [39], which we extensively evaluate in this paper. In a similar spirit, SlowFuzz [47] finds pathologically slow inputs, but uses a one-dimensional objective (i.e., the instruction count) which was proven to be less effective than PerfFuzz [39]. Finally, Blair proposed HotFuzz [16], a framework based on micro-fuzzing to find algorithmic complexity attacks in Java libraries.

## 7 Conclusions

Despite their popularity and broad application, regexps are still extremely difficult for users to get right. In particular, users may inadvertently expose themselves to ReDoS: a subtle, but deadly attack that can effectively stop all program progress. In this paper, we introduce REGULATOR, a novel dynamic analysis tool for finding ReDoS-vulnerable regexps. REGULATOR uses a novel approach: by instrumenting the real regexp matching system directly, it is able to effectively identify ReDoS without requiring complex analyses or extra effort to support modern regexp features. Moreover, REGULATOR can handle by design any additional features that will be added to the matching system in the future. We use REGULATOR to instrument IRREGEXP, one of the most popular regexp matching systems in use today. We find that REGULATOR is able to identify between two to seven times more vulnerable regexps than current state-of-the-art tools.

## 8 Acknowledgements

We would like to thank our reviewers for their valuable comments and input to improve our paper. This material is based upon work supported by the NSF under Award No. CNS-1704253, by the Office of Naval Research under Award No. N00014-17-1-2011, and by DARPA under agreement number HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the named funding agencies.

## References

[1] CVE-2021-23425. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23425.

[2] libFuzzer – a library for coverage-guided fuzz testing. https://www.llvm.org/docs/LibFuzzer.html.

[3] RegExr: Learn, Build, & Test RegEx. https://regexr.com/.

[4] CodeQL for research | GitHub Security Lab. https://securitylab.github.com/tools/codeql/, 2021.

[5] Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. https://github.com/google/re2, 2021.

[6] regex - rust. https://docs.rs/regex/1.5.4/regex/, 2021.

[7] regexlib.com. https://regexlib.com, 2021.

[8] Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/, 2021.

[9] StackOverflow. https://stackoverflow.com/, 2021.

[10] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata. In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, pages 108–120, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Jeff Avallone. Regexper. https://regexper.com.

[12] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 197–208. IEEE Press, 2019.

[13] Gina R Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T Stolee. Exploring tools and strategies used during regular expression composition tasks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 197–208. IEEE, 2019.

[14] Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Žd'árek, editors, *Proceedings of the Prague Stringology Conference 2017*, pages 30–41, Czech Technical University in Prague, Czech Republic, 2017.

[15] Joao Bispo, Ioannis Sourdis, Joao M.P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *2006 IEEE International Conference on Field Programmable Technology*, pages 119–126, 2006.

[16] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided microfuzzing. *arXiv preprint arXiv:2002.03416*, 2020.

[17] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[18] Mathias Bynens. ECMAScript regular expressions are getting better! https://web.archive.org/web/20210304234832/https://mathiasbynens.be/notes/es-regexp-proposals, 2017.

[19] Carl Chapman and Kathryn T. Stolee. Corpus dataset. https://github.com/softwarekitty/tour_de_source/blob/master/analysis/pattern_tracking/corpusPatterns.txt, 2015.

[20] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 282–293, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 405–416, 2017.

[22] Carl Chapman, Peipei Wang, and Kathryn T Stolee. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 405–416. IEEE, 2017.

[23] Nariyoshi Chida and Tachio Terauchi. Automatic repair of vulnerable regular expressions. *arXiv preprint arXiv:2010.12450*, 2020.

[24] Miles Claver, Jordan Schmerge, Jackson Garner, Jake Vossen, and Jedidiah McClurg. Regis: Regular expression simplification via rewrite-guided synthesis. *arXiv preprint arXiv:2104.12039*, 2021.

[25] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1280–1287, 2017.

[26] Erik Corry, Christian Plesner Hansen, and Lasse Reichstein Holst Nielsen. Irregexp, google chrome's new regexp implementation. =https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html, Feb 2009.

[27] James Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[28] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 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*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[29] James C Davis, Louis G Michael IV, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 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*, pages 443–454, 2019.

[30] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 343–359, Baltimore, MD, August 2018. USENIX Association.

[31] Firas Dib. regex 101: build, test, and debug regex. https://regex101.com/.

[32] Anonymzed for submission. Npm dataset. Anonymzed for submission., 2021.

[33] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[34] Seymour Ginsburg and Joseph Ullian. Ambiguity in context free languages. *Journal of the ACM*, 13(1):62–89, Jan 1966.

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

[36] James Halliday. safe-regex. https://github.com/substack/safe-regex/commits/master, 2013.

[37] Iain Ireland. A New RegExp Engine in SpiderMonkey. https://hacks.mozilla.org/2020/06/a-new-regexp-engine-in-spidermonkey, Jun 2020.

[38] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, pages 135–148, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[39] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. Flashregex: Deducing anti-redos regexes from examples. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 659–671. IEEE, 2020.

[41] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1063–1079. IEEE Computer Society, 2021.

[42] Louis G Michael, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 415–426. IEEE, 2019.

[43] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[44] Michael Mulder and George S Nezlek. Creating protein sequence patterns using efficient regular expressions in bioinformatics research. In *28th International Conference on Information Technology Interfaces, 2006.*, pages 207–212. IEEE, 2006.

[45] Andres Ojamaa and Karl Düüna. Security assessment of node.js platform. In Venkat Venkatakrishnan and Diganta Goswami, editors, *Information Systems Security*, pages 35–43, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[46] Jignesh Patel, Alex X. Liu, and Eric Torng. Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems. In *In Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[47] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[49] Asiri Rathnayake and Hayo Thielecke. Regexlib dataset. https://github.com/superhuman/rxxr2/blob/master/data/input/regexlib-raw.txt, 2016.

[50] Asiri Rathnayake and Hayo Thielecke. Snort dataset. https://github.com/superhuman/rxxr2/blob/master/data/input/snort-raw.txt, 2016.

[51] Lise Rommel Romero Navarrete and Guilherme P. Telles. Practical regular expression constrained sequence alignment. *Theoretical Computer Science*, 815:95–108, 2020.

[52] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: Crafting regular expression dos attacks. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 225–235. IEEE, 2018.

[53] Michael Sipser. *Introduction to the theory of computation*. Thomson Course Technology, Boston, 2nd ed.. edition, 2006.

[54] Henry Spencer. *A Regular-Expression Matcher*, page 35–71. Academic Press Professional, Inc., USA, 1994.

[55] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, page 20–26, New York, NY, USA, 2012. Association for Computing Machinery.

[56] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 361–376, 2018.

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

[58] Bryan Sullivan. New Tool: SDL Regex Fuzzer. https://www.microsoft.com/security/blog/2010/10/12/new-tool-sdl-regex-fuzzer/, 2010.

[59] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.

[60] Brink Van Der Merwe, Nicolaas Weideman, and Martin Berglund. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, pages 1–10, 2017.

[61] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. An empirical study on regular expression bugs. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 103–113, 2020.

[62] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. Demystifying regular expression bugs: A comprehensive study on regular expression bug causes, fixes, and testing. *arXiv preprint arXiv:2104.09693*, 2021.

[63] Peipei Wang and Kathryn T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 668–678, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Nicolaas Hendrik Weideman. *Static analysis of regular expressions*. PhD thesis, Stellenbosch: Stellenbosch University, 2017.

[65] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–20, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[66] Michał Zalewski. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt.