ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection

Yeting Li
SKLCS, ISCAS
UCAS

Zixuan Chen
SKLCS, ISCAS
UCAS

Jialun Cao
HKUST

Zhiwu Xu
Shenzhen University

Qiancheng Peng
SKLCS, ISCAS
UCAS

Haiming Chen ✉
SKLCS, ISCAS

Liyuan Chen
Tencent

Shing-Chi Cheung
HKUST

Abstract

Regular expression Denial of Service (ReDoS) is a class of algorithmic complexity attacks using the regular expressions (regexes) that cause the typical backtracking-based matching algorithms to run super-linear time. Due to the wide adoption of regexes in computation, ReDoS poses a pervasive and serious security threat. Early detection of ReDoS-vulnerable regexes in software is thus vital. Existing detection approaches mainly fall into two categories: static and dynamic analysis. However, they all suffer from either poor precision or poor recall in the detection of vulnerable regexes. The problem of accurately detecting vulnerable regexes at high precision and high recall remains unsolved. Furthermore, we observed that many ReDoS-vulnerable regex contain more than one vulnerability in reality. Another problem with existing approaches is that they are incapable of detecting multiple vulnerabilities in one regex.

To address these two problems, we propose ReDoSHunter, a ReDoS-vulnerable regex detection framework that can effectively pinpoint the multiple vulnerabilities in a vulnerable regex, and generate examples of attack-triggering strings. ReDoSHunter is driven by five vulnerability patterns derived from massive vulnerable regexes. Besides pinpointing vulnerabilities, ReDoSHunter can assess the degree (i.e., exponential or polynomial) of the vulnerabilities detected. Our experiment results show that ReDoSHunter achieves 100% precision and 100% recall in the detection of ReDoS-vulnerable regexes in three large-scale datasets with 37,651 regexes. It significantly outperforms seven state-of-the-art techniques. ReDoSHunter uncovered 28 new ReDoS-vulnerabilities in 26 well-maintained popular projects, resulting in 26 assigned CVEs and 2 fixes.

1 Introduction

Regular expressions (*regexes*) have wide applications in programming languages, string processing, database query languages and so on [1, 9, 14, 15, 20, 37]. Therefore, regexes are commonly used by online and offline services/projects for essential operations such as data validation, parsing, scraping and syntax highlighting [37, 44]. Earlier studies [8, 14] have reported that about 40% Java, JavaScript and Python projects use regexes. While regexes are popular, their computation can be complex and not easy to reason about. As a result, users or even experts often write regexes in super-linear worst-case time complexity (e.g., matching a string in quadratic or exponential time with the length of the input string). For example, `(\w|\d)+$` is a problematic regex commonly used to match strings ending with words or numeric characters. To determine whether a string w matches the regex, $O(2^{|w|})$ time may be needed due to backtracking. Furthermore, according to the recent investigations [14, 20], more than 10% of regexes used in software projects exhibit super-linear worst-case behavior.

More seriously, such regexes are subject to the Regular expression Denial of Service (abbrev., ReDoS, a.k.a. catastrophic backtracking) attacks. The threat of ReDoS is widespread and serious [14, 20, 40], and has a growing trend in recent years[1]. For instance, Stack Overflow [41] had a global outage in 2016 caused by a single super-linear regex. Similarly, in 2019, ReDoS took down Cloudflare's services [4]. Thus, early detection of ReDoS-vulnerable regexes in software projects is vital. Similar concerns are raised by Staicu and Pradel [40]: "*better tools and approaches should be created to help maintainers reason about ReDoS-vulnerabilities*".

Existing approaches for ReDoS-vulnerable regex identification are mainly either static or dynamic. However, existing detection approaches mostly involve a trade-off between precision and recall — a higher precision is often accompanied by a lower recall and vice versa. According to our investigation, the existing static work [14] with the highest recall (36.70%) turns out to result in only 57.96% precision. While the dynamic work [37] with 100% precision, results in only 1.82% recall. The huge trade-off on precision and recall limits

[1] Snyk's Security Research Team [39] found that there were a growing number of ReDoS-vulnerabilities disclosed, with a spike of 143% in 2018 alone.

the usefulness of these approaches. How to reach both high precision and high recall is still an open problem. Furthermore, the existing works can hardly locate the root cause of a ReDoS-vulnerability. Even the root cause of the vulnerability can be located, they can only detect one vulnerability. Nevertheless, according to our statistics (see §4.2), there are 53.7% of ReDoS-vulnerable regexes containing more than one vulnerability. This motivates the need for a ReDoS detection approach that can detect multiple vulnerabilities in a regex.

To achieve the end, we propose ReDoSHunter, a ReDoS-vulnerable regex detection framework, which can pinpoint multiple root causes of vulnerabilities in a regex and generate attack-triggering strings accordingly. Specifically, ReDoSHunter first adopts static analysis to identify potential vulnerabilities and generate attack strings that trigger the targeting vulnerabilities. The analysis leverages the five vulnerability patterns that we conclude by close examination of massive ReDoS-vulnerable regexes. These patterns prescribe the time complexity (exponential or polynomial), triggering conditions and possible attack strings (see §3.3 for details). Then, ReDoSHunter verifies whether the identified candidates are real vulnerabilities by dynamic analysis. Finally, ReDoSHunter outputs all the detected vulnerabilities with the degree (exponential or polynomial) and attack-triggering strings if any.

Empowered by the combination of static and dynamic analysis, and especially by the effectiveness of the patterns of the ReDoS-vulnerabilities, ReDoSHunter achieves high precision and recall at the same time. Our experiments show that ReDoSHunter achieves 100% precision and 100% recall on three large-scale datasets with 37,651 regexes. Furthermore, to validate the effectiveness of ReDoSHunter in the wild, we utilized ReDoSHunter to detect the publicly-confirmed real vulnerabilities in Common Vulnerabilities and Exposure (CVE) [12]. The experiment result shows ReDoSHunter can detect 100% of them, compared with the highest 60.00% achieved by the existing works. We applied ReDoSHunter to 26 well-maintained libraries (such as the popular JavaScript utility library lodash[2] which has more than 40 million weekly downloads), disclosing 28 new vulnerabilities among which 26 were assigned CVE IDs and 2 were fixed by developers.

The main contributions of this work are summarized as follows.

- We propose ReDoSHunter, a ReDoS-vulnerable regex detection framework which can pinpoint multiple root causes of vulnerabilities and generate attack-triggering strings. Combining both static and dynamic analyses, ReDoSHunter achieved remarkable precision and recall, reaching both 100% over three large-scale datasets, overcoming the dilemma as to which metric should be prioritized faced by the existing works.
- We identify five patterns of ReDoS-vulnerabilities based on extensive examination of massive vulnerable regexes. These patterns are characterized by detailed descriptions, degree of the vulnerability (the time complexity is exponential or polynomial), and the triggering conditions. They can help maintainers to locate ReDoS-vulnerabilities, shedding light on preventing and repairing vulnerable regexes.
- The experiment results demonstrate the practicality of ReDoSHunter. ReDoSHunter can detect 100% confirmed ReDoS-related CVEs, compared with the highest 60.00% achieved by the state-of-the-art works, and further identified 28 more unrevealed ReDoS-vulnerabilities across 26 intensively-tested projects, with 26 of them assigned CVEs and 2 of them fixed.

2 Preliminaries

Let Σ be an alphabet of all printable symbols except that each of the following symbols is written with an escape character \ in front of it: (,), {, }, [,], ^, $, |, \, ., ?, *, and +. Meanwhile, Σ also includes some special characters such as \t (denotes a tab character) and \n (denotes a newline character). The set of all words over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\varnothing$, respectively.

Definition 1. Standard Regular Expression. ε, $\varnothing$, and $a \in \Sigma$ are standard regular expressions; a standard regular expression is also formed using the operators: $\mathtt{r}_1|\mathtt{r}_2, \mathtt{r}_1\mathtt{r}_2, \mathtt{r}_1\{\mathtt{m,n}\}$, where $\mathtt{m} \in \mathbb{N}$, $\mathtt{n} \in \mathbb{N} \cup \{\infty\}$, and $\mathtt{m} \leq \mathtt{n}$. Besides, $\mathtt{r?}$, $\mathtt{r*}$, $\mathtt{r+}$ and $\mathtt{r}\{\mathtt{i}\}$ where $\mathtt{i} \in \mathbb{N}$ are abbreviations of $\mathtt{r}\{\mathtt{0,1}\}$, $\mathtt{r}\{\mathtt{0},\infty\}$, $\mathtt{r}\{\mathtt{1},\infty\}$ and $\mathtt{r}\{\mathtt{i,i}\}$, respectively. $\mathtt{r}\{\mathtt{m},\infty\}$ is often simplified as $\mathtt{r}\{\mathtt{m,}\}$.

The *language* $\mathcal{L}(\mathtt{r})$ of a standard regular expression $\mathtt{r}$ is defined inductively as follows: $\mathcal{L}(\varnothing) = \varnothing$; $\mathcal{L}(\varepsilon) = \{\varepsilon\}$; $\mathcal{L}(a) = \{a\}$; $\mathcal{L}(\mathtt{r}_1|\mathtt{r}_2) = \mathcal{L}(\mathtt{r}_1) \cup \mathcal{L}(\mathtt{r}_2)$; $\mathcal{L}(\mathtt{r}_1\mathtt{r}_2) = \{vw \mid v \in \mathcal{L}(\mathtt{r}_1), w \in \mathcal{L}(\mathtt{r}_2)\}$; $\mathcal{L}(\mathtt{r}\{\mathtt{m,n}\}) = \bigcup_{\mathtt{m} \leqslant i \leqslant \mathtt{n}} \mathcal{L}(\mathtt{r})^i$.

In practice, real-world regular expressions (regexes) are commonly found.

Definition 2. Real-world Regular Expression (regex). A regex over Σ is a well-formed parenthesized formula, consisting of operands in $\Sigma^* \cup \{\backslash\mathtt{i} \mid \mathtt{i} \geq 1\}$[3]. Besides the common rules governing standard regular expressions (e.g. $\mathtt{r}_1|\mathtt{r}_2, \mathtt{r}_1\mathtt{r}_2, \mathtt{r}_1\{\mathtt{m,n}\}$ defined in Definition 1), a regex also has the following constructs: (i) capturing group $(\mathtt{r})$; (ii) non-capturing group $(\mathtt{?:r})$; (iii) lookarounds: positive lookahead $\mathtt{r}_1(\mathtt{?=r}_2)$, negative lookahead $\mathtt{r}_1(\mathtt{?!r}_2)$, positive lookbehind $(\mathtt{?<=r}_2)\mathtt{r}_1$, and negative lookbehind $(\mathtt{?<!r}_2)\mathtt{r}_1$; (iv) anchors: Start-of-line anchor ^, End-of-line anchor $, word boundary \b, and non-word boundary \B; (v) lazy quantifiers: $\mathtt{r??}$, $\mathtt{r*?}$, $\mathtt{r+?}$, and $\mathtt{r}\{\mathtt{m,n}\}\mathtt{?}$; and (vi) backreference \i.

[2] https://www.npmjs.com/package/lodash

[3] In some environments, variables are used instead of \i.

A regex follows the syntactic rule that every control character `\i` is found to the right of the i-th capturing group, where capturing groups are indexed according to the occurrence sequence of their left parenthesis[4]. The same backreference can occur multiple times in a regex. In addition, the semantics of the constructs are explained in §3.2.

Regex matching is conducted with the support of a regex engine. Regex engines differ, but most (e.g., the built-in regex engines in Java and Python) will adopt *backtracking search* algorithms. Backtracking search algorithms can better support various grammatical extensions (e.g., lookarounds and backreferences). At the same time, they can also lead to potential Regular expression Denial of Service (ReDoS) attacks.

A regex `r` is *ReDoS-vulnerable* iff there exists a string w such that the regex on a backtracking regex engine has a super-linear behavior. Such strings are often called *attack strings*.

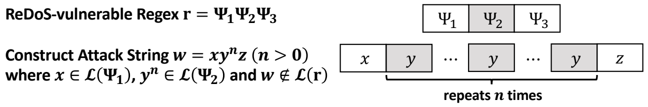

ReDoS-vulnerable Regex $\mathbf{r} = \Psi_1\Psi_2\Psi_3$
Construct Attack String $w = xy^nz$ $(n > 0)$
where $x \in \mathcal{L}(\Psi_1)$, $y^n \in \mathcal{L}(\Psi_2)$ and $w \notin \mathcal{L}(\mathbf{r})$
Ψ_1
Ψ_2
Ψ_3
x
y
...
y
...
y
z
repeats n times

Figure 1: The Components of the Attack String and the Relation Between the ReDoS-vulnerable Regex and the Attack string.

In our algorithms we find the position in the regex `r` that causes ReDoS, and locate a sub-regex containing this position, which is called the *infix* or *attackable* sub-regex of `r`. The sub-regexes before and after the infix sub-regex in `r` are called *prefix* and *suffix* sub-regexes, respectively. We use Ψ_1, Ψ_2, Ψ_3 to denote the prefix sub-regex, infix sub-regex, and suffix sub-regex respectively. Note that sub-regexes Ψ_1 and Ψ_3 can be ε. The components of the attack string $w = xy^nz$[5] and the relation between the ReDoS-vulnerable regex and the attack string is provided in Figure 1, which shows $w = xy^nz \notin \mathcal{L}(\mathtt{r})$, $n > 0$, $x \in \mathcal{L}(\Psi_1)$, and $y^n \in \mathcal{L}(\Psi_2)$. In addition, if $\Psi_1 = \varepsilon$, then $x = \varepsilon$.

For example, the regex $\Xi =$ `([0-9]*)+(\.[0-9]+)+` is ReDoS-vulnerable because the matching time of the regex Ξ on the Java-8 regex engine against a malicious string '`0`' × n grows rapidly with input size. (Figure 2)

For a standard regular expression r, the following sets are needed to analyze the *ambiguity* of r.

r.first $= \{a | au \in \mathcal{L}(r), a \in \Sigma, u \in \Sigma^*\}$;

r.last $= \{a | ua \in \mathcal{L}(r), a \in \Sigma, u \in \Sigma^*\}$;

r.followlast $= \{a | uav \in \mathcal{L}(r), u \in \mathcal{L}(r), u \neq \varepsilon, a \in \Sigma, v \in \Sigma^*\}$.

Consider the above regex Ξ, Ξ.first $= \Xi$.followlast $= \{$`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `\.`$\}$ and Ξ.last $= \{$`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`$\}$.

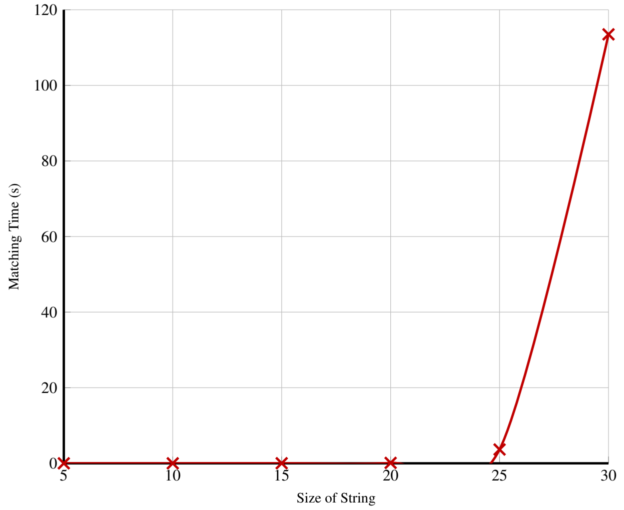


Figure 2: Matching Time against Malicious String Size for ReDoS-vulnerable Regex Ξ on the Java-8 Regex Engine.

We say r satisfies the *nullable* property if it accepts ε. We define r.nullable to represent this property as: r.nullable $=$ *true* if $\varepsilon \in \mathcal{L}(r)$ or *false* otherwise.

3 The ReDoSHunter Algorithm

In this section, we elaborate on the key ideas and techniques of our approach *ReDoSHunter* to analyze and identify the ReDoS-vulnerable regexes. Figure 3 shows the workflow of ReDoSHunter, which consists of three key components. The first component *regex standardization* (§3.2) transforms the original real-world regular expression (regex) into a simplified form which can then be manipulated by the second component. It takes a given regex as input and converts the regex into a standard regular expression with constraints using our designed transformation rules. The second component *static diagnosis* (§3.3) diagnoses the potential ReDoS-vulnerabilities of the given regex via the standard regular expression and the constraints obtained from the first component. In particular, it takes the standard regular expression and the constraints as input and diagnoses the potential backtracking locations, and then assesses the vulnerability degrees (exponential or polynomial) and generates the corresponding attack strings. The last component *dynamic validation* (§3.4) determines whether the candidate vulnerabilities diagnosed by the second component are real ones by testing and verifying these attack strings generated from the second component.

3.1 The Main Algorithm

Our algorithm ReDoSHunter is shown in Algorithm 1. ReDoSHunter first leverages the transformation rules that we design to rewrite the given regex α to a standard regular ex-

[4] Parentheses that are part of other syntax such as non-capturing groups should be skipped.

[5] It can also be denoted as $w = x + y \times n + z$,

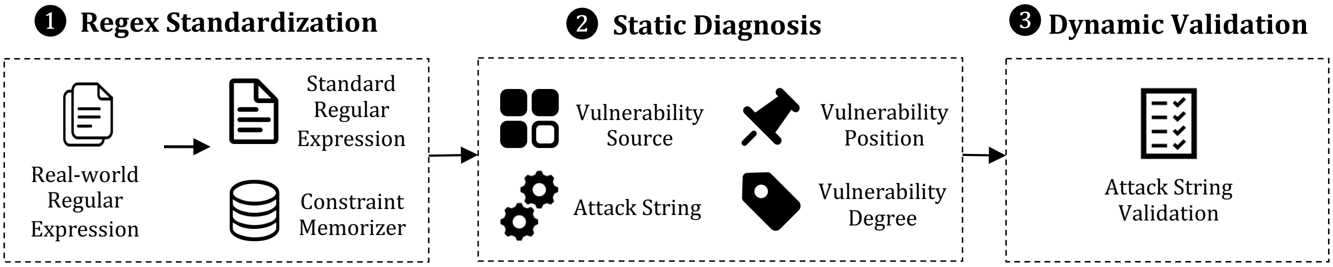


Figure 3: An Overview of ReDoSHunter for ReDoS Detection.

Table 1: The Vulnerability Type (Vuln. Type), Vulnerability Description (Vuln. Description), and Example Regex (including Attack String) of the Five ReDoS Patterns.

No.	ReDoS Pattern	Vuln. Type	Vuln. Description	Example Regex	Attack String
#1	Nested Quantifiers (NQ)	Exponential	Optional nested quantifiers result in two choices for each pump string	(CVE-2015-9239) \[(\d+;)?(\d+)*m	‘[’ + ‘0’ × 20 + ‘!’
#2	Exponential Overlapping Disjunction (EOD)	Exponential	A disjunction with a common outer quantifier whose multiple nodes overlap	(CVE-2020-7662) "((?:\\[\x00-\x7f]|[^\x00-\x08\x0a-\x1f\x7f"])*)"	‘"’ + ‘\\\x7e’ × 30 + ‘!’
#3	Exponential Overlapping Adjacency (EOA)	Exponential	Two overlapping nodes with a common outer quantifier {m,n} (n > 1) are either adjacent or can reach each other by skipping some optional nodes	(CVE-2018-3738) ^(?:\.?[a-zA-Z_][a-zA-Z_0-9]*)+$	‘a’ × 30 + ‘!’
#4	Polynomial Overlapping Adjacency (POA)	Polynomial	Two overlapping nodes with an optional common outer quantifier {0,1} are either adjacent or can reach each other by skipping some optional nodes	(CVE-2018-3737) ^([a-z0-9-]+)[␣\t]+([a-zA-Z0-9+\/]+[=]*)([\n␣\t]+([^\n]+))?$	‘0\t0’ + ‘\t’ × 10000 + ‘\n’
#5	Starting with Large Quantifier (SLQ)	Polynomial	The regex engine keeps moving the regex starting with a large quantifier across the string to find a match	(CVE-2019-1010266) [a-z][A-Z]|[A-Z]{2,}[a-z]|[0-9][a-zA-Z]|[a-zA-Z][0-9]|[^a-zA-Z0-9␣]	‘A’ × 10000 + ‘!’

```
Algorithm 1: ReDoSHunter
  Input: a regex $\alpha$
  Output: true, a diagnostic information list $\Gamma$ if $\alpha$ is
          ReDoS-vulnerable or false otherwise
1  $\beta$, $\mathcal{M} \leftarrow$ TransRE($\alpha$);
2  $\Gamma_{\mathcal{NQ}} \leftarrow$ CheckNQ($\beta$, $\mathcal{M}$);
3  $\Gamma_{\mathcal{EOD}} \leftarrow$ CheckEOD($\beta$, $\mathcal{M}$);
4  $\Gamma_{\mathcal{EOA}} \leftarrow$ CheckEOA($\beta$, $\mathcal{M}$);
5  $\Gamma_{\mathcal{POA}} \leftarrow$ CheckPOA($\beta$, $\mathcal{M}$);
6  $\Gamma_{\mathcal{SLQ}} \leftarrow$ CheckSLQ($\beta$, $\mathcal{M}$);
7  $\Gamma \leftarrow \Gamma_{\mathcal{NQ}} \cup \Gamma_{\mathcal{EOD}} \cup \Gamma_{\mathcal{EOA}} \cup \Gamma_{\mathcal{POA}} \cup \Gamma_{\mathcal{SLQ}}$;
8  if $|\Gamma| = 0$ then return false;
9  foreach info (vulDeg, vulSrc, vulPos, atkStr) $\in \Gamma$ do
10     if verifyAtk($\alpha$, atkStr, vulDeg) = false then
11         delete info (vulDeg, vulSrc, vulPos, atkStr) from
           $\Gamma$;
12 if $|\Gamma| > 0$ then return true, $\Gamma$;
13 else return false;
```

pression β with a constraint memorizer $\mathcal{M}$, which contains the constraints to generate attack strings that also belong to the original regex α (line 1). Next, according to β and $\mathcal{M}$, ReDoSHunter deduces the diagnostic information (i.e., *vulnerability degree*, *vulnerability source*, *vulnerability location*, and *attack string*) list Γ by statically detecting whether any of the five patterns (i.e., NQ, EOD, EOA, POA, and SLQ, as illustrated in Table 1), is triggered (lines 2-7). If Γ is empty, ReDoSHunter returns *false* (line 8), otherwise it dynamically verifies whether the attack strings in Γ are successful and the failed attack strings with their information are removed from Γ (lines 9-11). Finally, ReDoSHunter returns *true* and Γ if Γ is not empty (line 12), or returns *false* otherwise (line 13).

3.2 Regex Standardization

3.2.1 Extensions

As shown in §2, regexes support several useful extensions. We briefly explain them below.

A *lazy quantifier* is in the form of `r??`, `r*?`, `r+?` or `r{m,n}?`, which will match the shortest possible string. Match as few as possible, repeat as few times (i.e., the minimal number of times) as possible. *Anchors* do not match any characters, but still restrict the accepted words. The Start-of-line anchor `^` (resp. End-of-line anchor `$`) matches the position before the first (resp. after the last) character in the string. The word-boundary anchor `\b` can match the position where one side is a word and the other side is not a word. The anchor `\B` (non-word boundary) is a dual form of the word boundary `\b`. *Lookarounds* are useful to match something depending on the context before/after it. Specifically, a positive lookahead $\mathtt{r_1(?=r_2)}$ (resp. negative lookahead $\mathtt{r_1(?!r_2)}$) denotes looking for $\mathtt{r_1}$, but matching only if (resp. not) followed by $\mathtt{r_2}$. A positive lookbehind $\mathtt{(?<=r_2)r_1}$ (resp. negative lookbehind $\mathtt{(?<!r_2)r_1}$) means matching $\mathtt{r_1}$, but only if there's (resp. no) $\mathtt{r_2}$ before it. *Backreference* `\i` matches the exact same text that was matched by the *i*-th capturing group. A *capturing group* is enclosed in parentheses `(r)`. It enables us to get a portion of the match as a separate item in the result array. If we do not want a group to capture its match, we can optimize this group into `(?:r)` (i.e., *non-capturing group*).

3.2.2 Transformations

The high expressiveness of regexes makes many decision problems intractable or undecidable. To analyze ReDoS-vulnerabilities directly for regexes, we first convert a given regex into a standard regular expression with some constraints. The purpose is not to give an equivalent transformation, but instead trying to give a transformation with the same effect on ReDoS so that a source regex α has ReDoS-vulnerabilities iff the transformed target expression β has the same ReDoS-vulnerabilities. In case the "same effect" is hard to achieve, we seek a relaxed condition that allows the target expression to generate more ReDoS-vulnerabilities than the source regex. The dynamic validation guarantees reporting only true vulnerabilities. In the following, we introduce the transformations.

First, all the extensions are removed or changed. Specifically, the lazy quantifiers are changed to their eager forms by removing the '`?`'. This is based on our observation that if a regex with lazy quantifiers is ReDoS-vulnerable, the regex after removing lazy quantifiers is still ReDoS-vulnerable, and the two can be triggered by the same attack strings; while if the former is not ReDoS-vulnerable, the latter is neither. For similar reasons, we remove the non-capturing group sign '`?:`'. We also remove the Start-of-line anchor `^` (resp. End-of-line anchor `$`) to form a (possibly) relaxed regex as explained above.

For the regex $\mathtt{r_1\backslash br_2}$ with the word-boundary anchor `\b`, we convert the regex into an *over-approximated* expression $\mathtt{r_1r_2}$, and use the Constraint #1 in Table 2 to represent the regex $\mathtt{r_1\backslash br_2}$ equivalently. Similar conversions are made for `\B` and lookarounds.

Table 2: Constraint Generation.

No.	Extension	Constraints of Match String *w*
#1	$\mathtt{r_1\backslash br_2}$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(\mathtt{r_1}) \wedge w_2 \in \mathcal{L}(\mathtt{r_2}) \wedge \Big(\Big(\big(w_1 \in \mathcal{L}(\mathtt{.*\backslash W}) \vee w_1 = \varepsilon\big) \wedge w_2 \in \mathcal{L}(\mathtt{\backslash w.*})\Big) \vee \Big(w_1 \in \mathcal{L}(\mathtt{.*\backslash w}) \wedge \big(w_2 \in \mathcal{L}(\mathtt{\backslash W.*}) \vee w_2 = \varepsilon\big)\Big)\Big)$
#2	$\mathtt{r_1\backslash Br_2}$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(\mathtt{r_1}) \wedge w_2 \in \mathcal{L}(\mathtt{r_2}) \wedge \Big(\Big(\big(w_1 \notin \mathcal{L}(\mathtt{.*\backslash W}) \wedge w_1 \neq \varepsilon\big) \vee w_2 \notin \mathcal{L}(\mathtt{\backslash w.*})\Big) \wedge \Big(w_1 \notin \mathcal{L}(\mathtt{.*\backslash w}) \vee \big(w_2 \notin \mathcal{L}(\mathtt{\backslash W.*}) \wedge w_2 \neq \varepsilon\big)\Big)\Big)$
#3	$\mathtt{r_1(?=r_2)r_3}$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(\mathtt{r_1}) \wedge w_2 \in \mathcal{L}(\mathtt{r_2.*}) \wedge w_2 \in \mathcal{L}(\mathtt{r_3})$
#4	$\mathtt{r_1(?!r_2)r_3}$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(\mathtt{r_1}) \wedge w_2 \notin \mathcal{L}(\mathtt{r_2.*}) \wedge w_2 \in \mathcal{L}(\mathtt{r_3})$
#5	$\mathtt{r_1(?<=r_2)r_3}$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(\mathtt{r_1}) \wedge w_1 \in \mathcal{L}(\mathtt{.*r_2}) \wedge w_2 \in \mathcal{L}(\mathtt{r_3})$
#6	$\mathtt{r_1(?<!r_2)r_3}$	$w = w_1w_2$, where $w_1 \in \mathcal{L}(\mathtt{r_1}) \wedge w_1 \notin \mathcal{L}(\mathtt{.*r_2}) \wedge w_2 \in \mathcal{L}(\mathtt{r_3})$

For backreference `\i`, we first shape the regex with backreferences into an *over-approximated* backreference-free regex by adding an identifier $\lozenge_i$ after the referenced *i*-th capturing group, and replacing each backreference `\i` with the *i*-th capturing group with an identifier $\blacklozenge_i$ after it. We then memorize each identifier pair $\{\lozenge_i\colon \blacklozenge_i\}$ into the constraint memorizer, which requires the corresponding sub-regexes to match the same text. Note that the order of transforming the extensions is important. For example, transforming non-capturing groups should be made after transforming backreference.

Furthermore, several extensions need additional constraints as given in Table 2. Note that the constraints for backreference are given in the above. We use a constraint memorizer to record such information. After the above transformations, we obtain the target expression with a constraint memorizer. Note that for a regex without word-boundary anchors, lookarounds and backreferences, we do not return the constraint memorizer. No changes are made for regexes without any extension.

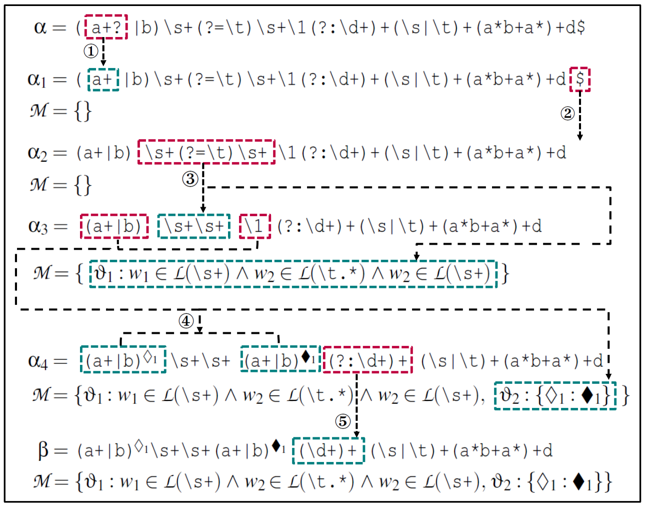

α = (a+?|b)\s+(?=\t)\s+\1(?:\d+)+(\s|\t)+(a*b+a*)+d$
①
α1 = (a+|b)\s+(?=\t)\s+\1(?:\d+)+(\s|\t)+(a*b+a*)+d$
M = {}
②
α2 = (a+|b)\s+(?=\t)\s+\1(?:\d+)+(\s|\t)+(a*b+a*)+d
M = {}
③
α3 = (a+|b) \s+\s+ \1 (?:\d+)+(\s|\t)+(a*b+a*)+d
M = {ϑ1 : w1 ∈ L(\s+) ∧ w2 ∈ L(\t.*) ∧ w2 ∈ L(\s+)}
④
α4 = (a+|b)◊1 \s+\s+ (a+|b)♦1 (?:\d+)+ (\s|\t)+(a*b+a*)+d
M = {ϑ1 : w1 ∈ L(\s+) ∧ w2 ∈ L(\t.*) ∧ w2 ∈ L(\s+), ϑ2 : {◊1 : ♦1}}
⑤
β = (a+|b)◊1\s+\s+(a+|b)♦1 (\d+)+ (\s|\t)+(a*b+a*)+d
M = {ϑ1 : w1 ∈ L(\s+) ∧ w2 ∈ L(\t.*) ∧ w2 ∈ L(\s+), ϑ2 : {◊1 : ♦1}}

Figure 4: The Transformations from Regex α to Regex β.

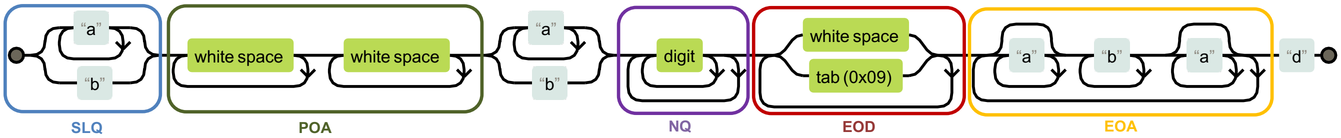


Figure 5: The Railroad Diagram of the ReDoS-vulnerable Regex β.

Now we use an example to illustrate the transformations, shown in Figure 4. Given a source regex α, TransRE first converts the regex α to the regex α_1 (see Figure 4) by changing the lazy quantifier `a+?` to `a+`. Then TransRE obtains the regex α_2 (shown in Figure 4) through deleting the anchor `$` directly. Next, TransRE first transforms the regex α_2 into the lookaround-free regex α_3 (given in Figure 4) via rewriting `\s+(?=\t)\s+` into `\s+\s+`, and puts the string constraint ϑ_1 (i.e., $w_1 \in \mathcal{L}(\texttt{\\s+}) \wedge w_2 \in \mathcal{L}(\texttt{\\t.*}) \wedge w_2 \in \mathcal{L}(\texttt{\\s+})$) into the memorizer $\mathcal{M}$. After that, TransRE adds an identifier $\Diamond_1$ after the 1st capture group (i.e., the sub-regex `(a+|b)`, called *parent sub-regex*) and rewrites the sub-regex `\1` (called *child sub-regex*) to the parent sub-regex with an identifier $\blacklozenge_1$ after it, which forms an over-approximate backreference-free regex α_4 (see Figure 4). And the identifier pair $\vartheta_2 = \{\Diamond_1\colon \blacklozenge_1\}$ is recorded into the memorizer $\mathcal{M}$. Finally, TransRE gets the target expression β (Figure 4) by removing the non-capturing group sign '`?:`'. Note that identifiers $\Diamond_1$ and $\blacklozenge_1$ only represent the marks of the 1st capture group and the backreference `\1`, and subsequent algorithms will not detect them as characters.

3.3 Static Diagnosis

In this section, we introduce five ReDoS patterns (i.e., NQ, EOD, EOA, POA, and SLQ) that are identified from our massive investigation and analysis. Among them, NQ, EOD and EOA have an exponential worst-case behavior on a mismatch (a.k.a. attack string), while POA and SLQ have a polynomial worst-case behavior. To identify these five patterns, we propose five static diagnosis algorithms, namely, CheckNQ, CheckEOD, CheckEOA, CheckPOA, and CheckSLQ. To leverage the performance and efficiency, these algorithms detect the necessary (but not necessarily sufficient) conditions to trigger the patterns. The vulnerability candidates detected are then dynamically validated such that only the true vulnerabilities are reported.

3.3.1 Pattern NQ: Nested Quantifiers

The first pattern concerns the expressions that have Nested Quantifiers (NQ). When matching a pump string, there are multiple possible choices among the nested quantifiers, leading to an exponential behavior in worst case on a mismatch. For example, the key portion `(\d+)*` in the real-world regex `\[(\d+;)?(\d+)*m` from CVE-2015-9239 meets the pattern NQ. And a pump string of a digit can be consumed by either the inner quantifier (`+`) or the outer one (`*`).

In order to diagnose the NQ pattern, we propose the algorithm CheckNQ. As shown above, the notable characteristic of the NQ pattern is the nested quantifiers. So CheckNQ first identifies all the NQ patterns in a transformed regex β by recursively checking whether each sub-regex β_1 has nested quantifiers.

Next, for each NQ pattern, based on the pattern and the constraints generated by the regex standardization, CheckNQ constructs a possible attack string, which is a candidate for dynamic validation. To be more precise, let us consider an NQ pattern β_1 in β, whose prefix and suffix sub-regexes are Ψ_1 and Ψ_3, respectively. Based on Ψ_1, β_1, and β, CheckNQ generates three strings x, y, and z such that (i) $x \in \mathcal{L}(\Psi_1)$, $y \in \mathcal{L}(\beta_1)$, $xyz \notin \mathcal{L}(\beta)$; and (ii) x, y, z satisfy the corresponding constraints in the memorizer $\mathcal{M}$ if exist (e.g., if Ψ_1 is transformed from `r`$_1$`\br`$_2$, then x should satisfy Constraint #1 in Table 2). CheckNQ then derives an attack string $w = x + y \times N_{\mathcal{E}} + z$, where $N_{\mathcal{E}}$ is a pre-defined number of repetitions for exponential patterns. Condition (i) guarantees the pump string w is a mismatch for the transformed expression, and $N_{\mathcal{E}}$ is to trigger an exponential behavior that can result in a lot of matching time. According to Condition (ii), w is a mismatch probably leading to an exponential behavior for the original regex α (suppose β is transformed from α).

CheckNQ also pinpoints the position of the NQ pattern in the original regex α according to the position of the NQ pattern in the regex β.

Let us consider the transformed expression in §3.2, that is $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d`. First, CheckNQ diagnoses that the sub-regex $\beta_1 =$ `(\d+)+` has nested quantifiers and thus is an NQ pattern, as illustrated in Figure 5. Then, CheckNQ extracts the prefix regex $\Psi_1 =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$ and tries to construct a prefix string x for it such that $x \in \mathcal{L}(\Psi_1)$. Moreover, x should also satisfy the corresponding constraints (i.e., ϑ_1 and ϑ_2 in §3.2) in the memorizer $\mathcal{M}$. So CheckNQ splits x into $x_1x_2x_3x_4$ such that $x_1 \in \mathcal{L}($`(a+|b)`$^{\Diamond_1})$, $x_2 \in \mathcal{L}(\texttt{\\s+})$, $x_3 \in \mathcal{L}(\texttt{\\s+})$, and $x_4 \in \mathcal{L}($`(a+|b)`$^{\blacklozenge_1})$. Note that the constraint $\vartheta_2 = \{\Diamond_1\colon \blacklozenge_1\}$ requires the two sub-regexes `(a+|b)`$^{\Diamond_1}$ and `(a+|b)`$^{\blacklozenge_1}$ should match the same text (i.e., $x_1 = x_4$), and the constraint ϑ_1 requires $x_3 \in \mathcal{L}(\texttt{\\t.*})$. So CheckNQ generates '`a`', '`\n`', '`\t`', '`a`' for x_1, x_2, x_3, x_4, respectively. Similarly, CheckNQ constructs an infix string $y =$ '`1`' and a suffix string $z =$ '`!`' such that $y \in \mathcal{L}($`(\d+)+`$)$ and $xyz \notin \mathcal{L}(\beta)$. Based on x, y, z, CheckNQ deduces an attack string $x + y \times N_{\mathcal{E}} + z$, where the repetition number $N_{\mathcal{E}}$ is set to 30. Finally, as the regex standardization does not change the relative positions of the sub-regexes in the given regex, CheckNQ can precisely locate the NQ pattern (?:\d+)+ in the original regex α according to the relative position of (\d+)+ in the transformed regex β.

3.3.2 Pattern EOD: Exponential Overlapping Disjunction

The second pattern is a disjunction with a common outer quantifier whose multiple disjuncts overlap, which is called Exponential Overlapping Disjunction pattern (EOD). When matching on a pump string, there are multiple possible choices among the overlapping disjuncts, leading to an exponential behavior in the worst case on a mismatch. Consider the expression (\w|\d)+$ shown in §1. As two disjuncts \w and \d overlap in the digits, when matching on a pump string of a digit, either \w or \d could be selected. Formally, an EOD is of the form $\beta = (\ldots(\beta_1|\beta_2|\ldots|\beta_k)\ldots)\{m_\beta, n_\beta\}$ with $n_\beta > 1$, satisfying one of the following conditions in Table 3. Intuitively, there is a string with multiple matching paths through alternation constructs in the pattern EOD.

Table 3: Conditions for Triggering Pattern EOD.

No.	Condition
#1	β_p.first $\cap$ β_q.first $\neq \varnothing$, where $1 \leq p, q \leq k$ and $p \neq q$
#2	β_p.first $\cap$ β_q.followlast $\neq \varnothing$, where $1 \leq p, q \leq k$ and $p \neq q$

We propose the algorithm CheckEOD to diagnose the EOD pattern. Like CheckNQ, CheckEOD consists of three steps: (i) identifying EOD patterns by the characteristics, (ii) constructing an attack string based on the pattern and the constraint memorizer $\mathcal{M}$, wherein the infix string y belongs to the overlapping part (i.e., a string with multiple matching paths), and (iii) locating the original source according to the relative positions of corresponding sub-regexes.

Consider the example mentioned in §3.2 again, $\beta =$ (a+|b)$^{\Diamond_1}$\s+\s+(a+|b)$^{\blacklozenge_1}$(\d+)+(\s|\t)+(a*b+a*)+d. First, as \s.first $\cap$ \t.first $= \{$\t$\} \neq \varnothing$, CheckEOD identifies the EOD pattern (\s|\t)+ (shown in Figure 5) and its prefix sub-regex $\Psi_1 =$ (a+|b)$^{\Diamond_1}$\s+\s+(a+|b)$^{\blacklozenge_1}$(\d+)+. Similar to §3.3.1, CheckEOD synthesizes the prefix string $x = x_1x_2x_3x_4x_5 =$ 'a\n\ta1', the infix string $y =$ '\t', and the suffix string $z =$ '!' such that $x \in \mathcal{L}(\Psi_1)$, $y \in (\mathcal{L}((\texttt{\\s+}) \cap \mathcal{L}((\texttt{\\t+}))\backslash\{\varepsilon\}$, $xyz \notin \mathcal{L}(\beta)$, and x satisfies the corresponding constraints (i.e., ϑ_1 and ϑ_2 in §3.2) in the memorizer $\mathcal{M}$. Next, based on x, y, z, an attack string $x + y \times N_{\mathcal{E}} + z$ is constructed, where the repetition number $N_{\mathcal{E}}$ is set to 30.

3.3.3 Pattern EOA: Exponential Overlapping Adjacent

The third pattern is an expression consisting of two adjacent overlapping components with a common outer quantifier {m,n}, where n > 1. We call it the Exponential Overlapping Adjacent pattern (EOA) as it could lead to an exponential behavior in the worst case on a mismatch. Specifically, there are two possible overlapping cases. First, the characters followed by the tail of the first component and the head ones of the second component overlap. For example, considering the regex (ab*b*)+, the characters following the tail (i.e., {b}) of the first component ab* and the head ones (i.e., {b}) of the second component b* overlap. When matching on the pump string of 'b', different components or paths can be selected. The common outer quantifier could make the matching an exponential behavior in the worst case. Second, the head characters of the first component and the ones following the tail of the second component overlap. Take the regex (a+b+a+)+ as an example. The head characters (i.e., {a}) of the first component a+ and the ones followed by the tail (i.e., {a}) of the second component b+a+ overlap. Due to the common outer quantifier +, the second component can reach the first component as well. Like the first case, matching on the pump string of 'b' could lead to an exponential behavior in the worse case.

Formally, the pattern EOA is of the form $\beta = (\ldots(\beta_1\beta_2)\ldots)\{m_\beta, n_\beta\}$ with $n_\beta > 1$, satisfying one of the conditions in Table 4. Following the cases of CheckNQ and CheckEOD, we propose the algorithm CheckEOA to detect the pattern EOA. Note that there may be more than one condition that are triggered by a regex (e.g., (a*a*)* triggers both of the above conditions). This has no effect on the detection of EOA, because we are concerned about whether the regex belongs to EOA, rather than about which form of EOA.

Table 4: Conditions for Triggering Pattern EOA.

No.	Condition
#1	(β_1.followlast $\cup$ β_1.last) $\cap$ β_2.first $\neq \varnothing$
#2	β_1.first $\cap$ (β_2.followlast $\cup$ β_2.last) $\neq \varnothing$

To illustrate CheckEOA, consider the example β again, $\beta =$ (a+|b)$^{\Diamond_1}$\s+\s+(a+|b)$^{\blacklozenge_1}$(\d+)+(\s|\t)+(a*b+a*)+d. CheckEOA identifies the sub-regex (a*b+a*)+ (Figure 5), as it triggers the second condition a*b+.first $\cap$ (a*.followlast $\cup$ a*.last) $= \{$a$\} \neq \varnothing$, as well as its prefix sub-regex $\Psi_1 =$ (a+|b)$^{\Diamond_1}$\s+\s+(a+|b)$^{\blacklozenge_1}$(\d+)+(\s|\t)+. Then, similar to §3.3.2, CheckEOA synthesizes the prefix string $x = x_1x_2x_3x_4 x_5x_6 =$ 'a\n\ta1\t', the infix string $y = y_1y_2 =$ 'ba', and the suffix string $z =$ '!' such that $x \in \mathcal{L}(\Psi_1)$, $y \in \mathcal{L}($(a*b+a*)+$)$, $xyz \notin \mathcal{L}(\beta)$, $y_2 \in$ a*b+.first $\cap$ (a*.followlast $\cup$ a*.last), and x satisfies the corresponding constraints (i.e., ϑ_1 and ϑ_2 in §3.2) in the memorizer $\mathcal{M}$. Next, based on x, y, z, an attack string $x + y \times N_{\mathcal{E}} + z$ is constructed, where the repetition number

$N_{\mathcal{E}}$ is set to 30.

3.3.4 Pattern POA: Polynomial Overlapping Adjacent

The fourth pattern is an expression consisting of two adjacent components such that the characters followed by the tail of the first component and the head ones of the second component overlap. Similar to the first case of the pattern EOA, matching on the overlapping string could select either of the components. But different from the pattern EOA, the pattern POA has with an optional common outer quantifier `{0,1}`. The ambiguity of the pattern POA could lead to a polynomial behavior in the worse case. So we call this pattern as Polynomial Overlapping Adjacent pattern (POA). For example, consider the regex `\d+\.?\d+$`. The characters followed by the tail of the first component `\d+$` are the digits, which also appear in the head of the second component `\.?\d+$`. Due to the quantifier `+`, the first component `\d+` can reach itself. When matching on the pump string of a digit, different components can be selected.

Formally, the pattern POA is of the form $\beta = \beta_1\beta_2$ such that β_1.followlast $\cap$ β_2.first $\neq \varnothing$. Likewise, the algorithm CheckPOA is proposed to detect the pattern POA.

Likewise, let us consider the example $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d` to illustrate algorithm CheckPOA. CheckPOA diagnoses that the sub-regex `\s+\s+` belongs to the pattern POA as it satisfies the condition `\s+`.followlast $\cap$ `\s+`.first $= \{␣, \texttt{\textbackslash t}, \texttt{\textbackslash n}, \texttt{\textbackslash r}, \ldots\} \neq \varnothing$ (symbol ␣ presents a space character) for the two adjacent `\s+`, as illustrated in Figure 5. And its prefix sub-regex $\Psi_1 =$ `(a+|b)`$^{\Diamond_1}$ is also identified. Next, CheckPOA constructs the prefix string $x =$ '`a`', the infix string $y = y_1y_2 =$ '`\t\t`', and the suffix string $z =$ '`!`' such that $x \in \mathcal{L}(\Psi_1)$, $y \in \mathcal{L}(\texttt{\textbackslash s+\textbackslash s+})$, $xyz \notin \mathcal{L}(\beta)$, $y_1 = y_2 \in$ `\s+`.followlast $\cap$ `\s+`.first, and $y_2 \in \mathcal{L}(\texttt{\textbackslash t.*})$ (i.e., the constraint ϑ_1 in the memorizer $\mathcal{M}$). After that, CheckPOA crafts an attack string such that it does not match the regex: $x + y \times N_{\mathcal{P}} + z$, where the repetition number $N_{\mathcal{P}}$, a pre-defined number of repetitions for polynomial patterns, is set to 10000 here.

3.3.5 Pattern SLQ: Starting with Large Quantifier

The above four patterns are all due to some ambiguity during the matching. Yet, some unambiguous regexes can be vulnerable when they cause the regex engine to keep moving the matching regex across the malicious string that does not have a match for the regex. For example, consider a simplified version `\s+$` of the regex that causes the outage of Stack Overflow mentioned in §1 and an attack string '`\t`' × 10000 + '`!`'. The matching starts with the first '`\t`' and fails after $10,000$ steps, and then continues on the second '`\t`' and so on. Finally, it would take $10,000 + 9,999 + 9,998 + \ldots + 3 + 2 + 1 = 50,005,000$ steps to reject the attack string, that is, a polynomial behavior in the worst case on a mismatch. There are several possible forms that can cause this vulnerability, and we find that the vulnerable parts are all at the beginning of the regex and with a large quantifier (the repetitions are greater than a minimal number). So we group them in a pattern called Starting with Large Quantifier (SLQ).

Next, we describe four possible triggering conditions for the pattern SLQ, as shown in Table 5, where $n_\beta \geq n_{min}$, $1 \leq p, q \leq k$, $p \neq q$, $1 \leq \ell$, and n_{min} is a pre-defined number for the minimal repetitions. We present algorithm CheckSLQ to detect the pattern SLQ based on these four conditions.

Table 5: Conditions for Triggering Pattern SLQ.

No.	Condition
#1	starting with $\beta_1\{m_\beta, n_\beta\}$
#2	starting with $\beta_1\beta_2\{m_\beta, n_\beta\}$ such that $(\mathcal{L}(\beta_1) \cap \mathcal{L}(\beta_2\{m_\beta, n_\beta\}))\backslash\{\varepsilon\} \neq \varnothing$
#3	starting with $\beta_1(\gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k)\{m_\beta, n_\beta\}$ such that there exists a word $w = w_0w_1 \ldots w_\ell \in \mathcal{L}(\gamma_p\{m_\beta, n_\beta\})$, $w_1 \ldots w_\ell w_0 \in \mathcal{L}(\gamma_q\{m_\beta, n_\beta\})$, and $w_0 \in \mathcal{L}(\beta_1)$
#4	starting with $\beta_1(\gamma_1\gamma_2 \ldots \gamma_k)\{m_\beta, n_\beta\}$ such that all the $\gamma_1, \gamma_2, \ldots, \gamma_k$ are nullable, and there exists a word $w = w_0w_1 \ldots w_\ell \in \mathcal{L}(\gamma_p\{m_\beta, n_\beta\})$, $w_1 \ldots w_\ell w_0 \in \mathcal{L}(\gamma_q\{m_\beta, n_\beta\})$, and $w_0 \in \mathcal{L}(\beta_1)$

Let us further examine the above example $\beta =$ `(a+|b)`$^{\Diamond_1}$`\s+\s+(a+|b)`$^{\blacklozenge_1}$`(\d+)+(\s|\t)+(a*b+a*)+d` to illustrate algorithm CheckSLQ. CheckSLQ detects that β starts with the sub-regex `a+`, as shown in Figure 5, which triggers the first condition, and constructs the prefix string $x = \varepsilon$, the infix string $y =$ '`a`', and the suffix string $z =$ '`!`' such that $x \in \mathcal{L}(\varepsilon)$, $y \in \mathcal{L}(\texttt{a+})$, and $xyz \notin \mathcal{L}(\beta)$. After that, CheckSLQ generates the attack string $x + y \times N_{\mathcal{P}} + z$, where the repetition number $N_{\mathcal{P}}$ is set to 10000.

3.4 Dynamic Validation

The principles of dynamic validation[6] (i.e., the algorithm verifyAtk) are: (i) to measure the time t for the source regex α to match the attack string *atkStr*, (ii) to check whether the corresponding threshold $T_{\mathcal{P}}$ (for polynomial vulnerability) or $T_{\mathcal{E}}$ (for exponential vulnerability) is triggered according to the vulnerability degree *vulDeg*, that is, if $t > T_{\mathcal{P}}$ (or corresponding $t > T_{\mathcal{E}}$) is satisfied[7], then verifyAtk returns *true*, otherwise returns *false*.

The step of dynamic validation is to address two issues. First, the static analysis can produce false positives. Note that the five patterns (i.e., NQ, EOD, EOA, POA, and SLQ) proposed in §3.3 are necessary but not necessarily sufficient conditions for judging whether a regex α is ReDoS-vulnerable,

[6] Our dynamic validation phase supports testing on the built-in regex engines in Python 2/3, Java 7-15, Node.js 6-14. Here we choose to test on the built-in regex engine in Java-8.

[7] For more sufficient validation, verifyAtk stops when the threshold is reached.

and dynamic validation is required as a supplement. Second, the transformed expressions may have been relaxed, detecting more ReDoS-vulnerabilities than the source regexes. So if dynamic validation is missing, false positives may occur. In other words, our dynamic validation phase can guarantee that the verified regexes must be actually vulnerable. For example, the regex $\alpha =$ ab*bc triggers the POA pattern, thus static diagnosis will judge that α is a polynomial vulnerability of ReDoS, and generate an attack string $atkStr =$ 'a' + 'b' $\times N_{\mathcal{P}}$ + '!'. However, the attack string *atkStr* does not cause catastrophic backtracking, so α is not a real ReDoS-vulnerable regex. The time consumption of dynamic validation for failed attacking is acceptable. For the example $\alpha =$ ab*bc, it takes only 0.278ms for α to match $atkStr =$ 'a' + 'b' $\times N_{\mathcal{P}}$ + '!'. The time consumed is highly acceptable.

Coming back to the example in §3.2 and §3.3, at last, verifyAtk tests and verifies that the matching time of the five attack strings exceeds the corresponding thresholds. Therefore, ReDoSHunter diagnoses that the regex α is ReDoS-vulnerable, and then return the corresponding diagnosis information list Γ, as shown in Table 6.

Table 6: The Diagnostic Information List Γ Reported by ReDoSHunter.

No.	Pattern	Vuln. Degree	Vuln. Position	Attack String
#1	NQ	Exponential	(?:\d+)+	'a\n\ta' + '1' × 30 + '!'
#2	EOD	Exponential	(\s|\t)+	'a\n\ta1' + '\t' × 30 + '!'
#3	EOA	Exponential	(a*b+a*)+	'a\n\ta1\t' + 'ba' × 30 + '!'
#4	POA	Polynomial	\s+(?=\t)\s+	'a' + '\t\t' × 10000 + '!'
#5	SLQ	Polynomial	a+?	'a' × 10000 + '!'

4 Experiments

In the experiments, we evaluate ReDoSHunter by studying three research questions:

RQ1. How is the effectiveness and efficiency of ReDoSHunter on large-scale regex sets? A good ReDoS detection tool should be able to efficiently distinguish ReDoS-vulnerable regexes from ReDoS-free ones over a large amount of regexes. Thus, we compared ReDoSHunter with seven state-of-the-art baselines on three real-world datasets in terms of precision and recall. We show the impact of different regex engines on the effectiveness of ReDoS detection. We also show the prevalence of multiple ReDoS-vulnerabilities in real-world regexes. Furthermore, we evaluate the effectiveness of generated attack strings by means of their attack success rates. (§4.2)

RQ2. How is the effectiveness of ReDoSHunter on identifying known vulnerabilities? The Common Vulnerabilities and Exposures (CVE) system is a database related to information security, publishing the confirmed vulnerabilities on open-source projects and the found ReDoS-vulnerabilities. Thus we conducted experiment on the confirmed ReDoS-related CVEs to compare the capabilities of existing works and ReDoSHunter. (§4.3)

RQ3. How is the effectiveness of ReDoSHunter on exploring unknown vulnerabilities? On top of RQ2, we further explore the capability of ReDoSHunter on disclosing unknown ReDoS-vulnerable regexes in intensively-tested projects, and submitted the detected vulnerabilities to CVEs. (§4.4)

4.1 Experiment Setup

4.1.1 Benchmark Datasets

We evaluate ReDoSHunter on three types of datasets (i.e., *regex sets*, *known ReDoS-vulnerabilities*, and *intensively-tested projects*). For the *regex sets*, we collected 37,651 regexes from three widely-used libraries (i.e., Corpus [8], RegExLib[8], and Snort[9]) of regexes. The details can be found in Table 7. For the *known ReDoS-vulnerabilities*, we collected vulnerabilities from widely-used libraries with Common Vulnerabilities and Exposures (CVE) [12] identifiers. We extracted CVEs with keywords "ReDoS", or "regular expression denial of service" (48 records), then manually filtered out those without clear descriptions or sources, resulting in 35 CVEs in total. Table 10 shows the details, including their CVE IDs, source projects, and detection results by all detectors. For *intensively-tested projects*, we selected 26 popular projects on GitHub/npm/PyPI with millions of downloads, applicable in various daily scenarios such as parsing and validating color, URL, HTML, email and so on. Table 11 lists the details, including source projects, disclosure status, and detection results by all detectors.

Table 7: The Regex Sets for Evaluation.

Name	Number	Avg Len	Description
Corpus	13,597	33.97	Regexes from scraped Python projects
RegExLib	8,699	69.75	Online regex examples from regexlib.com
Snort	15,355	92.28	Regexes extracted from the Snort NIDS for inspecting IP packets
Total:	37,651		

[8] https://regexlib.com
[9] https://www.snort.org

Table 8: **Comparison of the Overall Effectiveness over Four Popular Regex Engines on the Benchmarks with 37,651 Regexes.** Columns in each sub-tables denote the number of true positives (TP), the number of false positives (FP), the number of false negatives (FP), precision (Prec), and recall (Rec). The Real Vulnerabilities entries give the number of regexes that can trigger ReDoS attacks on various engines. The number of vulnerabilities reported by each technique is given by the sum of its TP and FP.

Regex Engine	**Java-8**					**Java-13**					**Python-3.7**					**Node.js-14**				
Technique	**TP**	**FP**	**FN**	**Prec** (%)	**Rec** (%)	**TP**	**FP**	**FN**	**Prec** (%)	**Rec** (%)	**TP**	**FP**	**FN**	**Prec** (%)	**Rec** (%)	**TP**	**FP**	**FN**	**Prec** (%)	**Rec** (%)
RXXR2	224	5	10,121	97.82	2.17	216	13	10,032	94.32	2.11	213	16	9,594	93.01	2.17	219	10	9,427	95.63	2.27
Rexploiter	2,052	288	8,293	87.69	19.84	2,041	299	8,207	87.22	19.92	1,955	385	7,852	83.55	19.93	1,915	425	7,731	81.84	19.85
NFAA	975	13	9,370	98.68	9.42	968	20	9,280	97.98	9.45	857	131	8,950	86.74	8.74	842	146	8,804	85.22	8.73
safe-regex	3,760	2,348	6,585	61.56	36.35	3,715	2,393	6,533	60.82	36.25	3,586	2,522	6,221	58.71	36.57	3,540	2,568	6,106	57.96	36.70
Regexploit	1,051	2	9,294	99.81	10.16	1,051	2	9,197	99.81	10.26	1,044	9	8,763	99.15	10.65	1,032	21	8,614	98.01	10.70
SDL	112	0	10,233	100	1.08	108	4	10,140	96.43	1.05	98	14	9,709	87.50	1.00	102	10	9,544	91.07	1.06
ReScue	188	0	10,157	100	1.82	183	5	10,065	97.34	1.79	175	13	9,632	93.09	1.78	179	9	9,467	95.21	1.86
ReDoSHunter	**10,345**	**0**	**0**	**100**	**100**	**10,248**	**0**	**0**	**100**	**100**	**9,807**	**0**	**0**	**100**	**100**	**9,646**	**0**	**0**	**100**	**100**
Real Vulnerabilities	10,345					10,248					9,807					9,646				

4.1.2 Baselines

To evaluate the effectiveness and efficiency of ReDoSHunter, we selected seven approaches, falling into two paradigms, i.e., static analysis (RXXR2 [22, 36], Rexploiter [49], NFAA [47], safe-regex [14] and Regexploit [26]) and dynamic analysis (SDL [43] and ReScue [37]). These approaches were among the state-of-the-art approaches used in recent works for ReDoS-specific detection. These two paradigms have their own pros and cons. So we compared ReDoSHunter with both of them.

4.1.3 Evaluation Metrics

We measure effectiveness using the precision and recall of reported vulnerabilities.

- **Precision**: the proportion of true positives (TPs, real vulnerabilities) over the reported vulnerabilities (the sum of true positives and false positives (FPs)).
- **Recall**: the proportion of the true positives over all the real vulnerabilities (the sum of true positives and false negatives (FNs)).

4.1.4 Configuration

We implemented the prototype of ReDoSHunter in Java-8, which supports the regex engines of Python 2/3, Java 7-15, Node.js 6-14. Our experiments were run on a machine with 2.20 GHz Intel Xeon(R) Silver processor and 128G RAM, running Windows 10. We used the parameter configuration $N_{\mathcal{P}} = 30,000$, $N_{\mathcal{E}} = 100$, $T_{\mathcal{P}} = 1$s, $T_{\mathcal{E}} = 0.1$s, and $n_{min} = 100$ in our algorithms for all experiments. All baselines were configured in the same settings as reported in their original papers.

4.2 Results on Regex Benchmarks

In this section, we present the experiment results comparing the performance between ReDoSHunter and the seven baselines on the three benchmark datasets.

Since it is labor intensive to manually identify and confirmed the vulnerable ones from the 37,651 regexes, we employed ReDoSHunter and the seven baselines to do the first-round filtering, resulting in a set of candidates labeled by any of eight techniques as vulnerable[10]. Then three experts analyzed the candidates and identified the real vulnerable ones manually. For manual analysis, three experts were involved and checked the candidate regexes independently. Then they cross-checked and discussed the results until reaching an agreement. Besides, the dynamic tools also validated the labeled results to some degree. Table 8 gives the overall evaluation results on the three benchmark datasets.

4.2.1 Effectiveness

We evaluate the effectiveness in terms of TP, FP, FN, precision, and recall in the reported vulnerabilities by each technique. The result is shown in Table 8. The comparison is based on four regex engines (i.e., Java-8, Java-13, Python-3.7, Node.js-14). The results given by these four engines are largely similar. To avoid repetition, we discuss the results mainly based on the Java-8 engine below.

[10] Here, whether a tool detects multiple vulnerabilities or a single vulnerability in a regex, this regex will be recorded as a vulnerable one.

According to Table 8, ReDoSHunter outperforms all baseline techniques in precision and recall. It successfully reports all ReDoS-vulnerable regexes without any false positives. In comparison, safe-regex achieves the highest recall 36.35% among all baselines with 61.56% precision. While achieving 100% precision, SDL and ReScue sacrifice recall, with only 1.08% to 1.82%. The experiment results show that the seven baselines suffer from either low TPs or high FPs and FNs. Specifically, the most TP achieved by others (3,760 achieved by safe-regex) is at most one third of that of ReDoSHunter (10,345), while the number of FNs of all existing works are relatively high, up to 10,233, as compared with the no FN achieved by ReDoSHunter. In terms of FP, the number of FPs of baselines range from 0 to 2,348, with an average of 379.42 ((5 + 288 + 13 + 2348 + 2 + 0 + 0) / 7 = 379.42). Some baselines have no FP at the cost of many FNs, resulting in poor recall (e.g., 1.08% achieved by SDL). The experiment result shows that ReDoSHunter can precisely detect far more vulnerabilities than any baselines.

In addition, we analyze the vulnerabilities commonly detected by each tool to further evaluate the effectiveness of ReDoSHunter. As shown in Figure 6 (Venn diagram), the amount of vulnerabilities detected by all baselines is still less than that of ReDoSHunter (in yellow). Besides, there are 4,487 ReDoS-vulnerabilities uniquely detected by ReDoSHunter, whereas no vulnerabilities can be uniquely detected by any baselines. These experiment results demonstrate that ReDosHunter is significantly more effective than all baselines in the detection of ReDoS vulnerabilities.

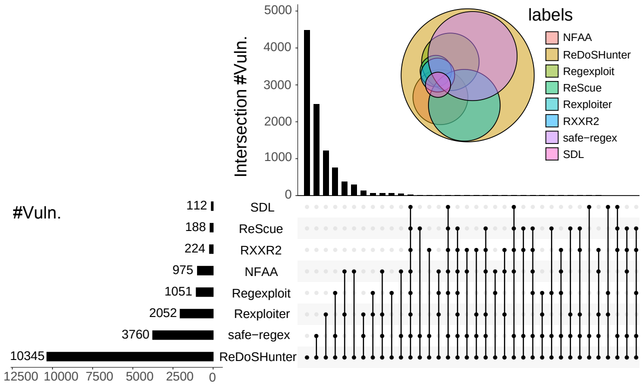


Figure 6: **An Illustration of Effectiveness of ReDoSHunter in Java-8.** The bar chart in the left-hand side shows the total number of ReDoS-vulnerabilities detected by each tool. The Venn diagrams illustrates the intersection of detected vulnerabilities of each tool. The two vertical bar charts together show the number of ReDoS-vulnerabilities (the upper vertical bar chart) that can be detected uniquely by the corresponding tools (the lower vertical bar chart with lined dots to represent the use of the corresponding tools).

4.2.2 Evaluation on Different Regex Engines

Some regexes may not be ReDoS-vulnerable on specific engines with particular implementation optimizations. Let us consider the regex mentioned in §2 again `([0-9]*)+(\.[0-9]+)+`, it is ReDoS-vulnerable in Python 2/3, Java-8 and Node.js 4-16, but not in Java-13. Therefore, we use multiple regex engines for TP, FP, FN, precision, and recall evaluation as shown in Table 8.

The precision and recall of baselines vary across regex engines. For example, NFAA achieves a 98.68% precision on the Java-8 engine but 85.22% precision on the Node.js-14 engine. In constrast, ReDoSHunter uniformly achieves 100% precision and 100% recall across the engines. The result demonstrates that ReDoSHunter's dynamic validation step can work well across different popular regex engines (Python 2/3, Java 7-15, Node.js 6-14). Furthermore, the numbers of vulnerabilities in different regex engines also indicate the performance difference of these engines varies. The more is the number of vulnerabilities detected, the less is the regex engine optimized. In our experiments, all eight techniques detected the most vulnerabilities on Java-8 (compared to Java-13, Python-3.7 and Node.js-14). In other words, the Java-8 regex engine is the least optimized, whose behavior resembles the Java-13/Python-3.7/Node.js-14 regex engine.

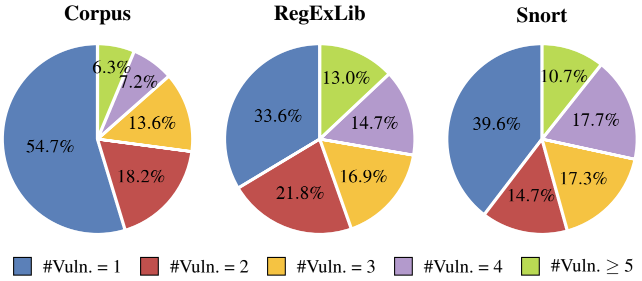


Figure 7: The Distribution Pie Charts of the Number of ReDoS-Vulnerabilities in Regexes Detected by ReDoSHunter in Java-8.

4.2.3 Multiple Vulnerabilities in One Regex

According to our observation, in one ReDoS-vulnerable regex, it has a high probability of containing more than one vulnerability. Figure 7 summarizes the percentage of different number of vulnerabilities (denoted as #Vuln.) in ReDoS-vulnerable regexes which were detected by ReDoSHunter. As we can see from Figure 7, there are more than a half (i.e., 66.4% and 60.4%) vulnerable regexes from RegExLib and Snort datasets containing more than one ReDoS-vulnerabilities — the amount of such regexes is non-negligible. However, the existing tools neglect this situation and thus are inapplicable to detect the multi-vulnerabilities in one regex, making it very likely to have serious consequences by reporting only one vulnerability. The result indicates a urgent need for effective tools that can identify multiple ReDoS-vulnerabilities from a vulnerable regex, reflecting the usefulness of ReDoSHunter.

4.2.4 Efficiency

The efficiency of different tools is illustrated in Figure 8. The left-hand side of the figure shows the average time of processing a regex, and right-hand side shows the number of ReDoS-vulnerable regexes detected within the runtime showed in the left-hand side. Overall, the static methods are much faster than dynamic methods, with less than one second per regex and about one minute per regex, respectively. And on average, ReDoSHunter has a comparable running time with static-based methods, taking around one second (1.06 seconds) to process one regex, and apparently outperforms the dynamic methods (up to 54.15 seconds). Considering the far more vulnerabilities detected by ReDoSHunter than existing methods, it is clear that ReDoSHunter is quite efficient.

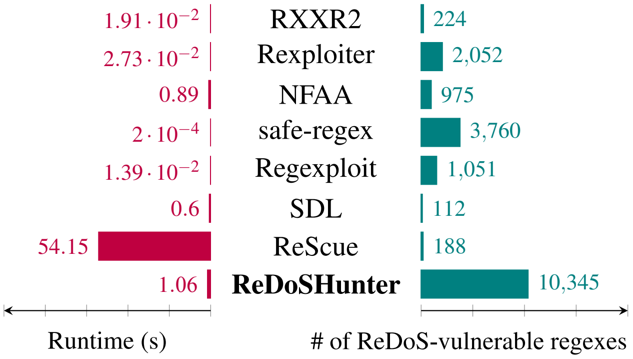


Figure 8: Comparison on the Running Time and the Number of ReDoS-vulnerable Regexes Detected.

4.2.5 Effectiveness of Generated Attack Strings

We further evaluate the effectiveness of attack strings generated by each tool in terms of the success attack rate (the number of strings that launch ReDoS attack successfully over the number of corresponding TPs). The result is shown in Table 9. Taking Java-8 as an example, we can see that the attack strings generated by the existing works are not always effective — the success attack rate ranges from 56.25% to 96.81%, none of them achieve 100%. The lowest success rate (56.25% achieved by SDL) is about half of ReDoSHunter (100%). For example, for the ReDoS-vulnerable regex `^("(\\["\\]|[^"])*"|[^\n])*$` from RegExLib, the tool RXXR2 generated a failed attack string ‘`""`’ × n (i.e., the matching time of the attack string is very fast, e.g., when $n = 30,000$, it only took 0.002s). Similar situation happens using different regex engines. Note that comparing with the unstable success rates achieved by other works (the most appearant difference is a decrease from 96.81% to 33.88% when changing regex engines from Java-8 to Java-13), ReDoSHunter provides attack strings with a stable 100% success rate, indicating the attack strings generated by ReDoSHunter are more effective than existing works.

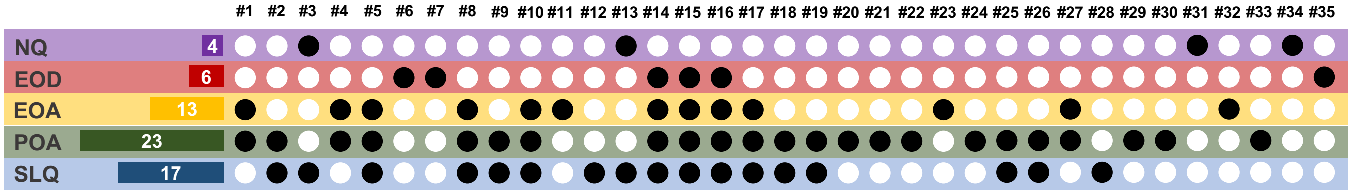
Table 9: **The Effectiveness of Generated Attack Strings.** The division formula represents the number of strings that successfully launch the ReDoS attack divided by the corresponding TPs. The symbol "–" indicates that the corresponding tools do not generate attack strings.

Technique	Java-8	Java-13	Python-3.7	Node.js-14
RXXR2	152 / 224 (67.86%)	114 / 216 (52.78%)	129 / 213 (60.56%)	142 / 219 (64.84%)
Reploiter	—	—	—	—
NFAA	724 / 975 (74.26%)	731 / 968 (75.52%)	519 / 857 (60.56%)	496 / 842 (58.91%)
safe-regex	—	—	—	—
Regexploit	989 / 1,051 (94.10%)	949 / 1,051 (90.29%)	984 / 1,044 (94.25%)	944 / 1,032 (91.47%)
SDL	63 / 112 (56.25%)	13 / 108 (12.04%)	54 / 98 (55.10%)	46 / 102 (45.10%)
ReScue	182 / 188 (96.81%)	62 / 183 (33.88%)	162 / 175 (92.57%)	150 / 179 (83.80%)
ReDoSHunter	**10,345 / 10,345** **(100.00%)**	**10,248 / 10,248** **(100.00%)**	**9,807 / 9,807** **(100.00%)**	**9,646 / 9,646** **(100.00%)**

Summary to RQ1: ReDoSHunter can achieve 100% precision and 100% recall against four tested regex engines, compared with the best dynamic approach reaching 100% precision yet only 1.82% recall, and the highest recall of static methods is only 36.35%. Also, the regexes with more than one ReDoS-vulnerabilities are prevalent, taking up to more than 60% in the collected datasets. Besides, all the attack strings generated by ReDoSHunter can launch ReDoS attack successfully, while none of existing works achieve 100% success rate. To sum up, ReDoSHunter achieved a remarkable balance between effectiveness and efficiency empowered by the advantages of both static and dynamic methods.

4.3 Results on Known Vulnerabilities

In this section, we exercise the existing approaches as well as ReDoSHunter against the confirmed ReDoS-related CVEs to show the effectiveness on identifying the real-world vulnerabilities. The result is shown in Table 10. The columns denote the source projects where the CVEs from (Project), the CVE index (CVE ID) and whether the approaches successfully identify the corresponding CVEs. We can see that



		#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	#17	#18	#19	#20	#21	#22	#23	#24	#25	#26	#27	#28	#29	#30	#31	#32	#33	#34	#35
NQ	4	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○
EOD	6	○	○	○	○	○	●	●	○	○	○	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
EOA	13	●	○	○	●	●	○	○	●	○	●	●	○	○	●	●	●	●	○	○	○	○	○	●	○	○	○	●	○	○	○	○	●	○	○	○
POA	23	●	●	○	●	●	○	○	●	●	●	○	○	○	●	●	●	●	●	●	●	●	●	○	●	●	●	●	○	●	●	○	○	●	○	○
SLQ	17	○	●	●	○	●	○	○	●	●	●	○	●	●	●	●	●	●	●	●	○	○	○	○	○	●	●	○	●	○	○	○	○	○	○	○

Figure 9: The Percentage of Five Patterns on Real-world ReDoS-vulnerabilities.

Table 10: **The Overall Evaluation Results on Real-world ReDoS-vulnerabilities.** The abbreviations RXR, RER, NAA, SAX, RET, SDL, RSE and RHR represent RXXR2, Rexploiter, NFAA, safe–regex, Regexploit, SDL, ReScue and ReDoSHunter, respectively. ✓ and ✗ denote whether the corresponding method can successfully detect the vulnerability or not.

No.	Project	CVE ID	RXR	RER	NAA	SAX	RET	SDL	RSE	RHR
#1	jquery-validation	CVE-2021-21252	✗	✗	✗	✓	✗	✗	✓	✓
#2	CairoSVG	CVE-2021-21236	✗	✓	✓	✗	✓	✗	✗	✓
#3	date-and-time	CVE-2020-26289	✗	✗	✗	✓	✗	✗	✓	✓
#4	fast-csv	CVE-2020-26256	✓	✗	✓	✓	✗	✗	✓	✓
#5	Python	CVE-2020-8492	✓	✓	✗	✓	✓	✗	✗	✓
#6	websocket-extensions	CVE-2020-7663	✓	✗	✗	✗	✗	✗	✓	✓
#7	websocket-extensions	CVE-2020-7662	✓	✗	✗	✗	✗	✗	✓	✓
#8	url-regex	CVE-2020-7661	✗	✓	✗	✓	✗	✗	✓	✓
#9	uap-core	CVE-2020-5243	✗	✗	✗	✗	✓	✗	✗	✓
#10	waitress	CVE-2020-5236	✗	✗	✗	✓	✓	✗	✗	✓
#11	Cisco IOS	CVE-2020-3408	✓	✓	✓	✓	✓	✓	✓	✓
#12	lodash	CVE-2019-1010266	✗	✗	✗	✗	✗	✗	✗	✓
#13	remarkable	CVE-2019-12041	✓	✗	✓	✓	✓	✓	✓	✓
#14	owasp-modsecurity-crs	CVE-2019-11391	✗	✗	✗	✓	✓	✗	✗	✓
#15	owasp-modsecurity-crs	CVE-2019-11390	✗	✗	✗	✓	✓	✗	✗	✓
#16	owasp-modsecurity-crs	CVE-2019-11389	✗	✗	✗	✓	✓	✗	✗	✓
#17	owasp-modsecurity-crs	CVE-2019-11388	✗	✗	✗	✓	✗	✗	✗	✓
#18	owasp-modsecurity-crs	CVE-2019-11387	✗	✗	✗	✗	✗	✗	✗	✓
#19	highcharts	CVE-2018-20801	✗	✗	✗	✗	✗	✗	✗	✓
#20	uap-core	CVE-2018-20164	✗	✓	✗	✓	✓	✗	✓	✓
#21	js-bson	CVE-2018-13863	✗	✗	✗	✓	✗	✗	✗	✓
#22	nodejs	CVE-2018-7158	✗	✗	✗	✗	✗	✗	✗	✓
#23	protobuf.js	CVE-2018-3738	✓	✓	✗	✓	✓	✓	✗	✓
#24	node-sshpk	CVE-2018-3737	✗	✓	✗	✓	✗	✗	✗	✓
#25	Python	CVE-2018-1061	✗	✓	✓	✗	✗	✗	✗	✓
#26	Python	CVE-2018-1060	✗	✗	✓	✗	✗	✗	✗	✓
#27	brace-expansion	CVE-2017-18077	✗	✗	✗	✓	✗	✗	✗	✓
#28	parsejson	CVE-2017-16113	✗	✗	✗	✗	✗	✗	✗	✓
#29	charset	CVE-2017-16098	✗	✗	✗	✗	✗	✗	✗	✓
#30	tough-cookie	CVE-2017-15010	✗	✓	✗	✗	✗	✗	✗	✓
#31	jshamcrest	CVE-2016-10521	✓	✗	✓	✓	✗	✗	✓	✓
#32	jadedown	CVE-2016-10520	✓	✗	✓	✓	✗	✓	✓	✓
#33	moment	CVE-2016-4055	✗	✗	✓	✓	✗	✗	✗	✓
#34	ansi2html	CVE-2015-9239	✓	✗	✓	✓	✓	✓	✓	✓
#35	marked	CVE-2015-8854	✗	✗	✗	✗	✗	✗	✗	✓
		Total	10	9	10	21	12	5	12	**35**
		%	28.57	25.71	28.57	60.00	34.29	14.29	34.29	**100**

among all methods, only our ReDoSHunter (RHR) can identify all 35 ReDoS-vulnerabilities, while the best existing works (i.e., SAX) can only identify around half of them (21 / 35 = 60.00%). While identification rate of other works range from 14.3% (5 / 35) to 34.4% (12 / 35). In addition, it is noteworthy that there are 7 / 35 (20.00%) unique CVEs (#12, #18, #19, #22, #28, #29, and #35) that can only be identified by ReDoSHunter, indicating the limitation of existing works and reflecting the effectiveness of ReDoSHunter.

To get more insights, we illustrate the presence of patterns on all 35 ReDoS-related CVEs, as depicted in Figure 9. The solid circle denotes the pattern that has been identified in the corresponding CVE. We can see that every one of 35 CVEs involves at least one pattern, indicating the effectiveness of our five patterns and reasoning about the high recall of ReDoSHunter.

Furthermore, to demonstrate the usefulness of ReDoSHunter, we present the case of CVE #35 (i.e., CVE-2015-8854, given in Figure 10) in detail. ReDoSHunter successfully detected two vulnerabilities in the regex. Specifically, ReDoSHunter not only generated two attack strings (‘_’ + ‘__’ × 100 + ‘!’, and ‘*’ + ‘**’ × 100 + ‘!’), but also diagnosed corresponding two EOD patterns (`(?:__|[\s\S])+` and `(?:\*\*|[\s\S])+`) that lead to the two ReDoS-vulnerabilities. In comparison, the other seven detectors all failed to detect any vulnerability. Furthermore, the vulnerability discloser only found one vulnerability[11] (correspondingly, the project maintainers only fixed one vulnerability[12]). This reveals the capability of ReDoSHunter to find real-world vulnerabilities.

Summary to RQ2: ReDoSHunter can identify all 35 ReDoS-related CVEs, compared with the best work identifying only over 60.00% of them. Besides, there are 20.00% CVEs (7 over 35) can only be identified by ReDoSHunter, indicating the effectiveness of the patterns we concluded. Therefore, to answer RQ2, ReDoSHunter significantly outperforms other seven state-of-the-art methods in finding real-world known ReDoS-vulnerabilities.

[11] https://github.com/markedjs/marked/issues/497

[12] https://github.com/markedjs/marked/commit/a37bd643f05bf95ff18cafa2b06e7d741d2e2157

```
var inline = {
// sub-regex (?:__|[\s\S])+
// vulnerable to '_' + '__' × 100 + '!' trigger EOD
// sub-regex (?:\*\*|[\s\S])+
// vulnerable to '*' + '*' × 100 + '!' trigger EOD
...
em:/^\b_((?:__|[\s\S])+?)_\b|^\*((?:\*\*|[\s\S])+?)\*(?!\*)/,
...
};
```

Figure 10: The npm package marked (25.1k stars) is a markdown parser and compiler. The marked package before 0.3.4 allows attackers to cause a denial of service (CPU consumption) via unspecified vectors that trigger a ReDoS issue for the em inline rule.

Table 11: **The Overall Evaluation Results on Unknown ReDoS-vulnerabilities.** The abbreviations RXR, RER, NAA, SAX, RET, SDL, RSE and RHR represent RXXR2, Rexploiter, NFAA, safe–regex, Regexploit, SDL, ReScue and ReDoSHunter, respectively. ✓ and ✗ denote whether the corresponding method can successfully detect the vulnerability or not.

No.	Project	Status	RXR	RER	NAA	SAX	RET	SDL	RSE	RHR
#1	ua-parser-js	CVE-2020-7733	✗	✗	✗	✗	✗	✗	✗	✓
#2	trim	CVE-2020-7753	✗	✗	✗	✗	✗	✗	✗	✓
#3	npm-user-validate	CVE-2020-7754	✗	✗	✗	✗	✗	✗	✗	✓
#4	dat.gui	CVE-2020-7755	✗	✓	✗	✗	✓	✗	✗	✓
#5	codemirror-js	CVE-2020-7760	✓	✗	✓	✓	✗	✗	✗	✓
#6	@absolunet/kafe	CVE-2020-7761	✓	✓	✓	✓	✓	✓	✓	✓
#7	express-validators	CVE-2020-7767	✓	✗	✗	✓	✓	✗	✗	✓
#8	djvalidator	CVE-2020-7779	✓	✓	✓	✓	✓	✓	✓	✓
#9	ua-parser-js	CVE-2020-7793	✗	✗	✗	✗	✗	✗	✗	✓
#10	glob-parent	CVE-2020-28469	✗	✗	✗	✗	✓	✗	✗	✓
#11	jinja2	CVE-2020-28493	✗	✓	✓	✗	✗	✗	✗	✓
#12	three	CVE-2020-28496	✗	✓	✓	✗	✗	✗	✗	✓
#13	lodash	CVE-2020-28500	✗	✗	✗	✗	✗	✗	✗	✓
#14	py	CVE-2020-29651	✗	✗	✗	✗	✗	✗	✗	✓
#15	uap-core	CVE-2021-21317	✗	✗	✗	✗	✗	✗	✗	✓
#16	CKEditor 5	CVE-2021-21391	✗	✓	✓	✗	✗	✗	✗	✓
#17	prism	CVE-2021-23341	✗	✗	✗	✗	✗	✗	✗	✓
#18	path-parse	CVE-2021-23343	✗	✗	✗	✗	✗	✗	✗	✓
#19	html-parse-stringify	CVE-2021-23346	✓	✓	✗	✓	✓	✓	✗	✓
#20	jspdf	CVE-2021-23353	✓	✗	✓	✓	✗	✗	✗	✓
#21	printf	CVE-2021-23354	✗	✗	✗	✓	✗	✗	✗	✓
#22	hosted-git-info	CVE-2021-23362	✗	✗	✗	✓	✗	✗	✗	✓
#23	browserslist	CVE-2021-23364	✗	✓	✓	✗	✗	✗	✗	✓
#24	postcss	CVE-2021-23368	✗	✗	✓	✗	✗	✗	✓	✓
#25	postcss	CVE-2021-23382	✗	✗	✗	✗	✗	✗	✗	✓
#26	ssri	CVE-2021-27290	✓	✗	✓	✓	✓	✗	✓	✓
#27	Python	Fixed	✗	✗	✗	✗	✗	✗	✓	✓
#28	validator	Fixed	✗	✗	✗	✗	✗	✗	✓	✓
		Total	7	8	10	9	7	3	6	**28**
		%	25.00	28.57	35.71	32.14	25.00	10.71	21.43	**100**

4.4 Results on Unknown Vulnerabilities

On top of the remarkable result of ReDoSHunter of identifying known vulnerabilities, we then explore the effectiveness of ReDoSHunter in the wild and compare it with other works. Specifically, for 26 popular-used projects on GitHub, npm and PyPI, we apply ReDoSHunter to identify whether there are possible ReDoS-vulnerable regexes. Once ReDoSHunter detected a vulnerable regex, we then reported to the maintainers and submit to CVE to get confirmation. To speed up the disclosure and report process, we cooperated with Snyk [39], a security research team, who helped us to verify the reproducibility and severeness of the ReDoS-vulnerable regexes, contact the maintainers of corresponding projects and assign CVE IDs once confirmed by the maintainers.

The results are shown in Table 11. In total, ReDoSHunter detected 28 ReDoS-vulnerable regexes in these 26 projects, 26 of them were assigned CVE IDs, and 2 of them were fixed by the maintainers. We also applied other methods to explore these projects as well, the results were unsatisfactory, with at most 35.71% ReDoS-vulnerabilities (an average of 25.51%) can be detected, leaving about 64% vulnerabilities unrevealed. The results are in line with the previous findings in §4.2 and §4.3, revealing the effectiveness of ReDoSHunter in exploring unrevealed vulnerabilities.

Summary to RQ3: ReDoSHunter is capable to be applied to exploring unknown ReDoS-vulnerabilities in the wild. Among 28 identified vulnerabilities, 26 of them were assigned CVEs or 2 of them were fixed by maintainers. Compared with existing works among which the best method can only detect 35.71%, ReDoSHunter is more effective in finding unknown ReDoS-vulnerabilities in the real-world projects.

5 Related Work

Recently, there has been significant interests in automated techniques for detecting the algorithmic complexity vulnerabilities (ACV) [3, 5–7, 13, 23, 27, 28, 31, 33, 38, 46]. In this paper, we focus on automatic detection on Regular expression Denial of Service (ReDoS) [18, 21, 48], which is a class of ACV. In the following, we present the most related work in the detection and defending of ReDoS attacks.

5.1 ReDoS Detection

There are several works [22, 35–37, 42, 43, 47, 49] targeting at detecting potential ReDoS-vulnerabilities, which can be mainly classified into two paradigms: static analysis [22, 35, 36, 47, 49] and dynamic analysis [37, 42, 43], as we discuss below.

Static Analysis. Approaches [22, 35, 36, 47, 49] falling into this paradigm mainly detect ReDoS-vulnerabilities by transforming the regexes into their self-defined models, and identifying ReDoS-vulnerable constructs from the models statically. These approaches are known for high efficiency. RXXR2 [35, 36] is a static analysis tool extended from RXXR [22]. It transforms the given regex into their proposed power DFA, and searches the attack string on top of the power DFA. However, most of extensions (e.g., lookarounds, and backreferences) are not supported by RXXR2. Also regexes with polynomial ReDoS-vulnerabilities are beyond its scope, while most of the ReDoS-vulnerable regexes are polynomial in the wild [14]. These limitations make it less effective. Another approach, Rexploiter [49], detects ReDoS-vulnerable regexes by combining complexity analysis of NFAs with sanitization-aware taint analysis. Though it provides an extra function (i.e., excluding user-input uncontrolled regexes), it does not supports most of the extensions (e.g., lookarounds, backreferences, and non-capturing groups). The tool safe-regex [14] conducts detection by identifying whether the pattern NQ is triggered, or the number of Kleene-Star is greater than a preset threshold. Though such pattern matching approach runs efficiently, there are more ReDoS patterns which fall beyond its capability. On the other hand, NFAA [47] can support extensions like lookarounds and non-capturing groups, yet it fails to support backreferences.

Dynamic Analysis. Dynamic-based approaches [37, 42, 43] detect ReDoS-vulnerabilities at run time, usually known for high precision compared with static analysis. Most dynamic analysis tools use dynamic fuzzing, which constantly search time-consuming strings with an actual regex engine, and from these to infer the regex's worst-case time complexity. SDL [42, 43] detects ReDoS by testing the matching time of regexes against a range of randomly-generated strings. Yet it does not support most extensions (e.g., anchors `\b` and `\B`, lazy quantifiers, lookarounds, backreferences, etc), making it less capable. Instead of generating random strings, Rescue [37] is designed for searching time-consuming strings. Due to the enormous string search space, it can only identify exponential or higher polynomial ReDoS-vulnerabilities, but is unable to detect lower polynomial ReDoS-vulnerabilities or deeply hided ReDoS-vulnerabilities. On the other hand, the effectiveness of genetic searching is also affected by the initialization, making result unstable at each run. Moreover, these dynamic-based approaches output a random attack string that does not provide any insight into the root causes of the ReDoS-vulnerability.

5.2 ReDoS Prevention or Alleviation

Various techniques [2, 10, 11, 16, 17, 19, 24, 25, 29, 30, 32, 34, 44, 45, 50] have been proposed to prevent or alleviate ReDoS attacks either by equivalent/approximate regex transformation or regex matching speedup.

Equivalent/Approximate Regex Transformation. This series of works [10, 11, 24, 45] try to find equivalent/approximate ReDoS-invulnerable regexes to replace the ReDoS-vulnerable ones. Among them, Van der Merwe et al. [45] and Cody-Kenny et al. [11] devote to finding equivalent ReDoS-invulnerable regexes to replace the original ones. However, their use of exact equivalence is too strong in practice [14, 37], which limits their deployment to real-world applications. Chida and Terauchi [10], and Li et al. [24] address this problem by deducing anti-ReDoS regexes adopting programming-by-example algorithms. Yet the quality of anti-ReDoS regex deduced by them highly depends on the quality of user-provided examples.

Regex Matching Speedup. ReDoS attacks can also be alleviated by regex matching speedup, which is an alternative solution in some special cases, e.g., by parallel algorithms [25], GPU-based algorithms [50], state-merging algorithms [2], Parsing Expression Grammars (PEGs) [17, 19, 29], counting automata matching algorithm [44], memoization-based optimization [16] and recursion-limit/backtracking-limit/time-limit [30, 32, 34]. These works can alleviate ReDoS-vulnerability issues, yet they do not resolve the ReDoS-vulnerable regexes themselves, leaving them still subjecting to ReDoS attacks.

6 Discussion

Despite the remarkable effectiveness of ReDoSHunter, we notice there are still room for improvement. First, **Supports for more extensions**. ReDoSHunter can support most commonly-used extensions, while for those that are not commonly used such as conditional statements, ReDoSHunter does not consider them currently. However, they can be supported with suitable preprocessing. For example, for the regex with conditional statement $\texttt{(}r_1\texttt{)?(?(1)}r_2\texttt{|}r_3\texttt{)}$, it can be transformed to an over-approximate conditional statement-free regex and some external constraints so that ReDoSHunter can handle it. Second, **Supports for more characters.** Currently, ReDoSHunter supports common characters including unicode characters ranging from `U+0000`—`U+FFFF`, which can cover the most characters used in practice. While for characters falling beyond this range, ReDoSHunter may not detect them. This limitation can also be solved by an appropriate preprocess.

7 Conclusion

In this paper, we proposed ReDoSHunter, a ReDoS-vulnerable regex detection framework that can pinpoint multiple root causes of vulnerabilities and generate attack-triggering strings. It takes advantages of static and dynamic analysis, achieving a remarkable balance between precision and recall, reaching 100% precision and 100% recall over three large-scale datasets in the experiments. It successfully identified all the confirmed CVEs that are caused by ReDoS, and exposed 28 new ReDoS-vulnerabilities in popular open-source projects with 26 assigned CVEs and 2 fixed by the maintainers. We hope our work may provide insights of reasoning about the ReDoS-vulnerabilities, and shed lights on the automatic or semi-automatic ReDoS-vulnerable regex repair.

Acknowledgment

The authors would like to thank Adam Goldschmidt, Asaf Biton, Assaf Ben Josef, Benji Kalman, Colin Ife, George Gkitsas, Gur Shafriri, Hadas Bloom, Leeya Shaltiel, Sam Sanoop from Snyk Security Research Group for their great efforts on confirming and assigning CVEs. Also, the authors would like to thanks the anonymous reviewers for their helpful feedback. This work is supported in part by National Natural Science Foundation of China (Grants #61872339, #61472405, #61932021, #61972260, #61772347, #61836005), National Key Research and Development Program of China under Grant #2019YFE0198100, Guangdong Basic and Applied Basic Research Foundation under Grant #2019A1515011577, Huawei PhD Fellowship, and MSRA Collaborative Research Grant.

References

[1] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. Knowl. Data Eng.*, 28(5):1217–1230, 2016.

[2] Michela Becchi and Srihari Cadambi. Memory-Efficient Regular Expression Search Using State Merging. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*, pages 1064–1072. IEEE, 2007.

[3] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[4] The Cloudflare Blog. Details of the Cloudflare outage on July 2, 2019, 2020. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[5] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 463–473, 2009.

[6] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 27–41, 2009.

[7] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 186–199, 2009.

[8] 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, Saarbrücken, Germany, July 18-20, 2016*, pages 282–293, 2016.

[9] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring Regular Expression Comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 405–416, 2017.

[10] Nariyoshi Chida and Tachio Terauchi. Automatic Repair of Vulnerable Regular Expressions. *CoRR*, abs/2010.12450, 2020.

[11] 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, GECCO 2017, Berlin, Germany, July 15-19, 2017*, pages 1280–1287, 2017.

[12] The MITRE Corporation. Common Vulnerabilities and Exposures (CVE), 2020. https://cve.mitre.org/index.html.

[13] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[14] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations*

of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 246–256, 2018.

[15] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 443–454, 2019.

[16] James C. Davis, Francisco Servant, and Dongyoon Lee. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *2021 IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, May 23-27, 2021*, page To appear, 2021.

[17] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122, 2004.

[18] Jan Goyvaerts. Runaway Regular Expressions: Catastrophic Backtracking, 2020. `https://www.regular-expressions.info/catastrophic.html`.

[19] IBM. Rosie Pattern Language (RPL), 2020. `https://rosie-lang.org/`.

[20] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 415–426, 2019.

[21] Tim Kadlec. Regular Expression Denial of Service (ReDoS) and Catastrophic Backtracking, 2017. `https://snyk.io/blog/redos-and-catastrophic-backtracking/`.

[22] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, pages 135–148, 2013.

[23] 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, Amsterdam, The Netherlands, July 16-21, 2018*, pages 254–265, 2018.

[24] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. FlashRegex: Deducing Anti-ReDoS Regexes from Examples. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 659–671, 2020.

[25] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. Accelerating Regular Expression Matching Using Hierarchical Parallel Machines On GPU. In *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*, pages 1–5. IEEE, 2011.

[26] Doyensec LLC. Regexploit: DoS-able Regular Expressions, 2021. `https://github.com/doyensec/regexploit`.

[27] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. Symbolic Complexity Analysis Using Context-Preserving Histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 58–68, 2017.

[28] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. Complexity Vulnerability Analysis Using Symbolic Execution. *Softw. Test. Verification Reliab.*, 30(7-8), 2020.

[29] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From Regexes to Parsing Expression Grammars. *Sci. Comput. Program.*, 93:3–18, 2014.

[30] Microsoft. Regex class - C#, 2020. `https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=net-5.0`.

[31] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 322–332, 2018.

[32] PCRE. PCRE - Perl Compatible Regular Expressions, 2020. `https://pcre.org/`.

[33] 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 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2155–2168, 2017.

[34] PHP. PHP: preg_match - Manual, 2020. `https://www.php.net/manual/en/function.preg-match.php`.

[35] Asiri Rathnayake. *Semantics, Analysis And Security Of Backtracking Regular Expression Matchers*. PhD thesis, University of Birmingham, UK, 2015.

[36] Asiri Rathnayake and Hayo Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. *CoRR*, abs/1405.7058, 2014.

[37] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 225–235, 2018.

[38] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pages 89–98, 2006.

[39] Snyk. The state of open-source security, 2020. https://snyk.io/.

[40] 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 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 361–376, 2018.

[41] Stack Exchange Network Status. Outage Postmortem - July 20, 2016, 2020. https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[42] Bryan Sullivan. New Tool: SDL Regex Fuzzer, 2010. http://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer.

[43] Bryan Sullivan. Regular Expression Denial of Service Attacks and Defenses, 2010. https://docs.microsoft.com/en-us/archive/msdn-magazine/2010/may/security-briefs-regular-expression-denial-of-service-attacks-and-defenses.

[44] Lenka Turoňová, Lukás Holík, Ondrej Lengál, Olli Saarikivi, Margus Veanes, and Tomás Vojnar. Regex Matching with Counting-Set Automata. *Proc. ACM Program. Lang.*, 4(OOPSLA):218:1–218:30, 2020.

[45] 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, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017*, pages 38:1–38:10, 2017.

[46] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern Fuzzing for Worst Case Complexity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 213–223, 2018.

[47] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, pages 322–334, 2016.

[48] Adar Weidman. Regular Expression Denial of Service - ReDoS, 2017. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.

[49] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pages 3–20, 2017.

[50] Xiaodong Yu and Michela Becchi. GPU Acceleration Of Regular Expression Matching For Large Datasets: Exploring The Implementation Space. In Hubertus Franke, Alexander Heinecke, Krishna V. Palem, and Eli Upfal, editors, *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 18:1–18:10. ACM, 2013.