RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix

Yeting Li, *CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences; SKLCS, Institute of Software, Chinese Academy of Sciences;* Yecheng Sun, *SKLCS, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Zhiwu Xu, *College of Computer Science and Software Engineering, Shenzhen University;* Jialun Cao, *The Hong Kong University of Science and Technology;* Yuekang Li, *School of Computer Science and Engineering, Nanyang Technological University;* Rongchen Li, *SKLCS, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Haiming Chen, *SKLCS, Institute of Software, Chinese Academy of Sciences; CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences;* Shing-Chi Cheung, *The Hong Kong University of Science and Technology;* Yang Liu, *School of Computer Science and Engineering, Nanyang Technological University;* Yang Xiao, *CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity22/presentation/li-yeting

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

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix

Yeting Li[†§¶] Yecheng Sun[¶§] Zhiwu Xu[♮] Jialun Cao[‡] Yuekang Li[♯]
Rongchen Li[¶§] Haiming Chen[¶†✉] Shing-Chi Cheung[‡] Yang Liu[♯] Yang Xiao[†§]

[†]*CAS-KLONAT, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[§]*University of Chinese Academy of Sciences, Beijing, China*
[¶]*SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[♮]*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China*
[‡]*The Hong Kong University of Science and Technology, Hong Kong, China*
[♯]*School of Computer Science and Engineering, Nanyang Technological University, Singapore*

Abstract

The Regular expression Denial of Service (ReDoS) is a class of denial of service attacks that exploit vulnerable regular expressions (regexes) whose execution time can be superlinearly related to input sizes. A common approach of defending ReDoS attacks is to repair the vulnerable regexes. Techniques have been recently proposed to synthesize repaired regexes using program-by-example (PBE) techniques. However, these existing techniques may generate regexes, which are not semantically equivalent or similar to the original ones, or are still vulnerable to ReDoS attacks.

To address the challenges, we propose RegexScalpel, an automatic regex repair framework that adopts a *localize-and-fix* strategy. RegexScalpel first localizes the vulnerabilities by leveraging fine-grained vulnerability patterns proposed by us to analyze their vulnerable patterns, the source (i.e., the pathological sub-regexes), and the root causes (e.g., the overlapping sub-regexes). Then, RegexScalpel targets to fix the pathological sub-regexes according to our predefined repair patterns and the localized vulnerability information. Furthermore, our repair patterns ensure that the repair regexes are semantically either equivalent to or similar to the original ones. Our iterative repair method also keeps out vulnerabilities of the repaired regexes. With an experiment on a total number of 448 vulnerable regexes, we demonstrate that RegexScalpel can outperform all existing automatic regexes fixing techniques by fixing 348 more regexes than the best existing work. Also, we adopted RegexScalpel to detect ten popular projects including Python and NLTK, and revealed 16 vulnerable regexes. We then applied RegexScalpel to successfully repair all of them, and these repairs were merged into the later release by the maintainers, resulting in 8 confirmed CVEs.

1 Introduction

Regular expressions (*regexes*) are widely used in many computer science fields such as programming languages, string processing, database query, etc [1,6,11,12,18,33]. Attacks targeting regexes at online services are common [11,18,24,34]. These attacks are often launched by submitting a malicious input string to an online server and triggering the corresponding regex that requires a matching time of super-linear (i.e., polynomial or exponential) worst-case with respect to the input length, leading to retarding responses from the server. For example, millions of users were affected when Cloudflare [3] went down for 27 minutes in 2019. Such attacks are known as Regular expression Denial of Service (ReDoS) attacks. According to recent statistics [11,18], more than 10% of regexes used in software projects are vulnerable to ReDoS attacks.

There are three necessary conditions for successful ReDoS attacks: (i) a *slow regex engine*, (ii) an *attacker-controlled input string* and (iii) a *vulnerable regex* [11]. Practitioners have deployed different defense strategies to break one or more necessary conditions. To break the first condition, one strategy is to substitute a faster regex engine for a slower one (*i.e.*, *regex engine substitution*). This helps to improve the worst-case complexity of regexes by using alternative matching algorithms. However, such improvement is derived at the cost of omitting extended features (e.g., backreferences and lookarounds) [16] or space [13]. Besides, engine substitution can bring semantic differences [12] or incompatibilities [13]. A strategy to break the second condition is to limit the length of the input string (*i.e.*, *input length restriction*). But this strategy faces a dilemma known as "Goldilocks problem" [14], i.e., either the limited length is too short to accept all valid inputs, or the length is too long to reject malicious inputs. Furthermore, according to our experiment (see Sec. 5), the above defense methods ignore an important and universal vulnerable pattern, known as $\mathcal{SLQ}$ (See Sec. 4.1.4 or prior work [19]), which appears in 75.89% (340 / 448) of the benchmarks. As a result, these defense methods can only fix 13.82% (47 / 340) of the benchmarks, leaving 86.18% of them vulnerable to ReDoS attacks.

On searching for better defense strategies, practitioners find that repairing vulnerable regexes (*i.e.*, *regex repair*) can greatly mitigate their vulnerabilities by breaking the third condition. Our statistics in Table 13 show that regex repair is the most common defense strategy adopted by developers or maintainers, accounting for as high as 92.19%. However, regex repair is challenging. It is non-trivial to propose a semantic-equivalent or semantic-similar regex to substitute the vulnerable one, even for human experts. In addition, the proposed regexes can vary significantly across experts and newbies. In other words, manual regex repair is neither practical nor scalable. Moreover, there could be multiple vulnerabilities in a vulnerable regex. According to our investigation, 44.20% (198 / 448) vulnerable regexes have more than one vulnerability. Similar statistics were also observed by a recent work [19] published in 2021. The existence of multiple vulnerabilities in a vulnerable regex exacerbates the difficulty of automatic regex repair.

Current state-of-the-art techniques [7, 22] adopt programming-by-example (PBE) approaches to automatically repair ReDoS-vulnerable regexes. A vulnerable regex is repaired by synthesizing another regex that accepts the given positive examples (i.e., valid input strings) and rejects the given negative examples (e.g., ReDoS attacks). However, there are two drawbacks of the PBE approaches. **First, a synthesized regex may not be *semantically equivalent or similar* to its original regex with respect to all valid input strings.** The effectiveness of synthesis is limited by the insufficiency of high-quality examples [20, 25]. Insufficient or biased examples may result in the over-fitting or under-fitting of the synthesized regexes, leading to incorrect repairs. **Second, the repaired regex may be still vulnerable to ReDoS attacks.** These techniques ignore the $\mathcal{SLQ}$ vulnerable pattern, which leads to the incapability of repairing such vulnerable regexes. For example, Chida and Terauchi [7] return a repaired regex `<span ([^".1-8B-Y\[\\\]^b-dfh-y]*)font-style:italic ([^>]*)>` for the vulnerable regex `<span [^>]*font-style:italic[^>]*>`. The sub-regex `([^".1-8B-Y\[\\\]^b-dfh-y]*)` in the repaired regex is semantically very different from the pathological sub-regex `[^>]*` in the original one. As a result, the string `'<span class="replace-with italic">'` is matched by the original regex but not by the repaired one. In addition, the repaired regex contains an $\mathcal{SLQ}$ pattern, which can be attacked by some malicious attack strings, such as `'<spannfont-style:italic'.repeat(10000)`.

To overcome these challenges, we propose RegexScalpel, a regex ReDoS vulnerability analysis and repair framework based on localize-and-fix. RegexScalpel first leverages the fine-grained vulnerability patterns proposed in this paper, which enables analyzing the information necessary for the repair, to localize the vulnerabilities, including their vulnerable patterns, the source (i.e., the pathological sub-regexes), and the root causes (e.g., the overlapping sub-regexes). Then, RegexScalpel aims at fixing the pathological sub-regexes according to the predefined repair patterns as well as the localized vulnerability information. By removing the overlapping paths or reducing the maximum times of backtracking using micro-manipulations (e.g., adding a lookaround, deleting a quantifier or sub-regex, modifying a quantifier or sub-regex), the localized pathological sub-regexes are fixed and the ReDoS attack is thus defended. Furthermore, our repair patterns ensure that the repair regexes are semantically either equivalent to or similar as the original ones, and support comprehensive types of vulnerable causes including $\mathcal{SLQ}$. Our iterative repair method also keeps out vulnerabilities of the repaired regexes.

Our experiment shows the remarkable effectiveness of RegexScalpel. Among the 448 vulnerable regexes collected from practice, RegexScalpel can repair 443 (98.88%) of them. In contrast, the top performer of existing methods can only synthesize ReDoS-safe regexes for 95 (21.20%) of them. The experiment also reveals the limitation of manual repair – the maintainers may ignore repairing some vulnerabilities if there are many in one vulnerable regex. As a comparison, RegexScalpel repaired all vulnerabilities without manual effort and domain knowledge. Finally, we also applied RegexScalpel to real-world popular projects (including Python, NLTK, etc.) that are under repair. RegexScalpel successfully repaired 16 vulnerabilities, and the repaired regexes were confirmed by the developers and merged into subsequent releases.

In summary, this paper makes four major contributions:

- **Novelty.** We develop RegexScalpel, which is a ReDoS vulnerability analysis and repair framework. To the best of our knowledge, RegexScalpel is the first approach to localize and fix the vulnerable regexes considering comprehensive and fine-grained types of vulnerable causes.
- **Semantics Preservation.** We design a repair algorithm to preserve the semantics of the original regex. In particular, the language of the repaired regex is a subset of the language of the original one.
- **Effectiveness.** The evaluation exhibits the remarkable effectiveness of RegexScalpel. It achieves 98.88% successful repair ratio, compared with 21.20% achieved by the best existing work.
- **Usefulness.** We also employed RegexScalpel to help repair 16 vulnerable regexes across ten projects including Python and NLTK. The repaired results were merged into their later release.

2 Background

Before presenting RegexScalpel, we introduce the background of Regexes and ReDoS. Let Σ be a finite alphabet. The set of all words over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\emptyset$, respectively. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$.

2.1 Regular Expressions (Regexes)

A regex r over Σ is a well-formed parenthesized formula using capturing group, non-capturing group, named capturing group, backreference, greedy quantifier, lazy quantifier, lookarounds, and anchors, etc. Here, the regexes $r?$, $r*$, $r+$ and $r\{i\}$ where $i \in \mathbb{N}$ are abbreviations of $r\{0,1\}$, $r\{0,\infty\}$, $r\{1,\infty\}$ and $r\{i,i\}$, respectively. Besides, the regex $r\{m,\infty\}$ is often simplified as $r\{m,\}$.

Next, we explain the semantics of the regex constructs informally with the aid of JavaScript examples. A *capturing group* `(r)` allows getting a part of the match as a separate item. For example, `"aab!".match(/a+(b+)/)` returns `["aab", "b"]`, where `"aab"` is the match of the whole regex and `"b"` is the match of the capturing group `(b+)`[1]. If we do not want a group to capture its match, we can optimize this group into a *non-capturing group* `(?:r)`. For example, the matching result of `/a+(?:b+)/` is the same to `/a+b+/`. Moreover, we can access the result array by index. For example, `"aab!".match(/a+(b+)/)[1]` returns `"b"`. Some regex engines (*e.g.*, ES2018) also support a more convenient alternative to access capturing groups by names (i.e., *named capturing groups*). The named capturing groups are available in the property `groups` (e.g., `groups.day`, as seen in the following example).

```
let regex = /(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "10-16";
let groups = str.match(regex).groups;
console.log(groups.day); // 16
```

A *greedy quantifier* is repeated as many times as possible while a *lazy quantifier* is repeated as few times as possible. For example, `"123.456".match(/\d+\.\d+?/)` will return the array `["123.4"]`.

A *backreference* `\i` means "to match the same text as in the i-th capturing group". Similarly, to reference a named capturing group `(?<name>r)`, we can use backreference `\k<name>`. An example is shown in the following listing.

```
let str = "'foo'";
let r1 = /(['"])(.*?)\1/;
console.log(str.match(r1)); // ["'foo'", "'", "foo"]
let r2 = /(?<quote>['"])(.*?)\k<quote>/;
console.log(str.match(r2)); // ["'foo'", "'", "foo"]
```

Lookarounds are zero-length assertions, and search for strings that satisfy certain contexts. Specifically, it includes *lookahead* and *lookbehind*, specifying the context after and before the searching strings, respectively. A *positive lookahead* r_1`(?=`r_2`)` looks for a string that matches r_1 only when the string is followed by a string that matches r_2. Similarly, a *negative lookahead* r_1`(?!`r_2`)` searches for a string matching r_1 only when the string is not followed by a string that matches r_2. On the other hand, a *positive lookbehind* `(?<=`r_2`)`r_1 looks for a string which matches r_1 only when there is a string which matches r_2 before it. Similarly, a *negative lookbehind* `(?<!`r_2`)`r_1 looks for a string which matches r_1 only when the string before it does not match r_2. An example is shown in the following listing.

```
let str = "123foo456foz";
console.log(str.match(/[0-9]{3}(?=foo)/));  // ["123"]
console.log(str.match(/[0-9]{3}(?!foo)/));  // ["456"]
console.log(str.match(/(?<=foo)[0-9]{3}/)); // ["456"]
console.log(str.match(/(?<!foo)[0-9]{3}/)); // ["123"]
```

Anchors specify the non-character context. Specifically, a *start-of-line anchor* `^` denotes the start of a line, while an *end-of-line anchor* `$` denotes the end. A *word boundary anchor* `\b` matches the position where one side is a word and the other side is not a word. A *non-word boundary anchor* `\B`, the negation of `\b`, matches at any position between two word characters as well as at any position between two non-word characters. An example is given in the following listing.

```
let str = "123 4567";
console.log(str.match(/^[0-9]{3}/));   // ["123"]
console.log(str.match(/[0-9]{3}$/));   // ["567"]
console.log(str.match(/[0-9]{3}\b/)); // ["123"]
console.log(str.match(/[0-9]{3}\B/)); // ["456"]
```

Alphabet of a Regex. The *alphabet* Σ_r of a regex r is the set of all symbols that appear in r. For example, for the regex $r_1 =$ `(?=a)[ab]{1,3}`, the alphabet Σ_{r_1} is `{a,b}`.
Language of a Regex. The *language* $\mathcal{L}(r)$ of a regex r is the set of all words accepted by r. For the above regex r_1, the language $\mathcal{L}(r_1)$ is `{a,aa,ab,aaa,aab, aba,abb}`. Note that regexes with backreferences are not context-free [4]. We take inspiration from Li et al. [19] and use an *over-approximate* solution – calculate the language of i-th capturing group instead of calculating the language of backreference `\i`.

2.2 Regex Denial of Service (ReDoS)

The ReDoS is a type of algorithmic complexity attack. In a ReDoS attack, an attacker exploits a combination of a problematically ambiguous regex and a malicious input string to trigger the worst-case complexity of the regex engine, and thereby prevents legitimate users from accessing online services. An example is given below. Let us consider an ambiguous regex $r_2 =$ `/(a|a)+b/`. An attacker can craft the malicious input string $s = \underbrace{a\ldots a}_{k \text{ times}}$, which is not matched by the regex r_2, to make a backtracking-based regex engine explore all 2^k failing paths.

[1] In JavaScript, without the global search flag "g", the method *match*() returns the first match as an array, wherein the first element is the match of the whole regex (*i.e.*, the implicit capturing group 0) and the remaining are the matches of the capturing groups (if any exist) in order.

In this paper, we exploit the regex repair strategy to fix ReDoS vulnerabilities.

Regex Repair. Given a ReDoS-vulnerable regex r and a set of test cases $\mathcal{S}$, we aim to repair the regex r to form a new regex r' such that (i) r' passes all the given test cases $\mathcal{S}$; and (ii) r' is invulnerable to ReDoS attacks.

Semantics preservation. Semantics preservation after repair in this paper means that the language of the repaired regex is a subset of the language of the original one.

First Set. To fix ReDoS vulnerabilities, we introduce the *first*(r) set of a regex r:

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

Intuitively, first(r) represents the set of all symbols that may appear in the first position of the strings accepted by the regex r. Taking the above regex r_1 = `(?=a)[ab]{1,3}` as an example, first$(r_1) = \{$`a`$\}$.

3 Overview

In this section, we illustrate how our ReDoS-vulnerable regex repair framework RegexScalpel works using a motivating example.

The workflow consists of three key components, as shown in Figure 1. The first component *Analyzer* (§4.1) takes a given ReDoS-vulnerable regex as input, diagnoses the vulnerability source, then returns the vulnerability information including pattern (*i.e.*, the vulnerability type), source (*i.e.*, the pathological sub-regex), and cause (*e.g.*, the overlapping sub-regexes). The second component *Repairer* (§4.2) takes the vulnerability information as input and then repairs the ReDoS-vulnerable regex. The last component *Verifier* (§4.3) determines whether the repaired regexes are ReDoS-invulnerable and whether it can pass all the given test cases.

Example 1. Take the ReDoS-vulnerable regex `{([\s\S]+?)}` as an example. The regex aims to match the simple string formatting using `{}`, as mentioned in the NPM package nodejs-tmpl[2] (6,858,130 weekly downloads).

The first step of our approach is to leverage a vulnerability analyzer to diagnose the vulnerability source of the above ReDoS-vulnerable regex `{([\s\S]+?)}`. As shown in the phase ❶ of Figure 2, the analyzer identifies that the pathological sub-regex r=`{[\s\S]+`[3] is a (possible) vulnerability, due to its two components r_1 = `{` and r_2 = `[\s\S]+` overlap (i.e., $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) = \{$"{"$\} \neq \emptyset$), and returns a triple ($\mathcal{SLQ}_3$, r, $[r1, r2]$), where $\mathcal{SLQ}_3$ is a vulnerability pattern introduced in §4.1.4. In order to facilitate fixing the vulnerability, we also use r_p and r_q to represent the sub-regex of r_1 and r_2 without the outermost quantifier, that is, `{` and `[\s\S]`, respectively.

[2] https://github.com/daaku/nodejs-tmpl

[3] The capturing group and the lazy quantifier are ignored.

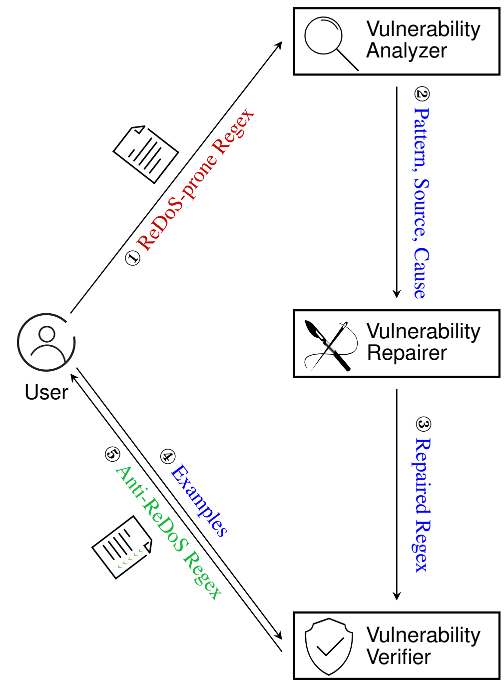

Figure 1: An Overview of RegexScalpel for ReDoS Repair.

A Proof of Concept (PoC) example of this vulnerability is shown in the following listing.

```
var tmpl = require("tmpl")
tmpl("{".repeat(50000)+"answer", { answer: "zxc" });
```

The second step of our approach is to repair the ReDoS-vulnerable regex, according to the vulnerability information obtained from the first step. The repaired regexes are also called patches. The key idea for repairing is to avoid catastrophic backtracking during matching. Considering the above vulnerability, we propose four possible repair patterns (see §4.2.4 for more detail) to fix it, as given in the phase ❷ of Figure 2. Specifically, we can add a start-of-line anchor ^before { (*i.e.*, the first repair pattern t_1 in Figure 2), yielding a patch γ_1 = `^{([\s\S]+?)}`. The start-of-line anchor ^ specifies that the following pattern r (i.e., `{([\s\S]+?)}`) must begin at the first position of the string, which can effectively avoid catastrophic backtracking (i.e., ReDoS). Similarly, we can also generate a patch γ_2 = `{([\s\S]{1,500}?)}` which substitutes the plus quantifier with a small quantifier $n_\mu = 500$ (*i.e.*, the second pattern t_2 in Figure 2). Obviously, the small quantifier $n_\mu = 500$ can cut the number of backtracking.

Eliminating the overlaps is a natural solution to avoid catastrophic backtracking. Therefore, we propose the third repair pattern t_3 with a negative lookahead `(?!)` in Figure 2 to make r_q should not be matched on the strings matched

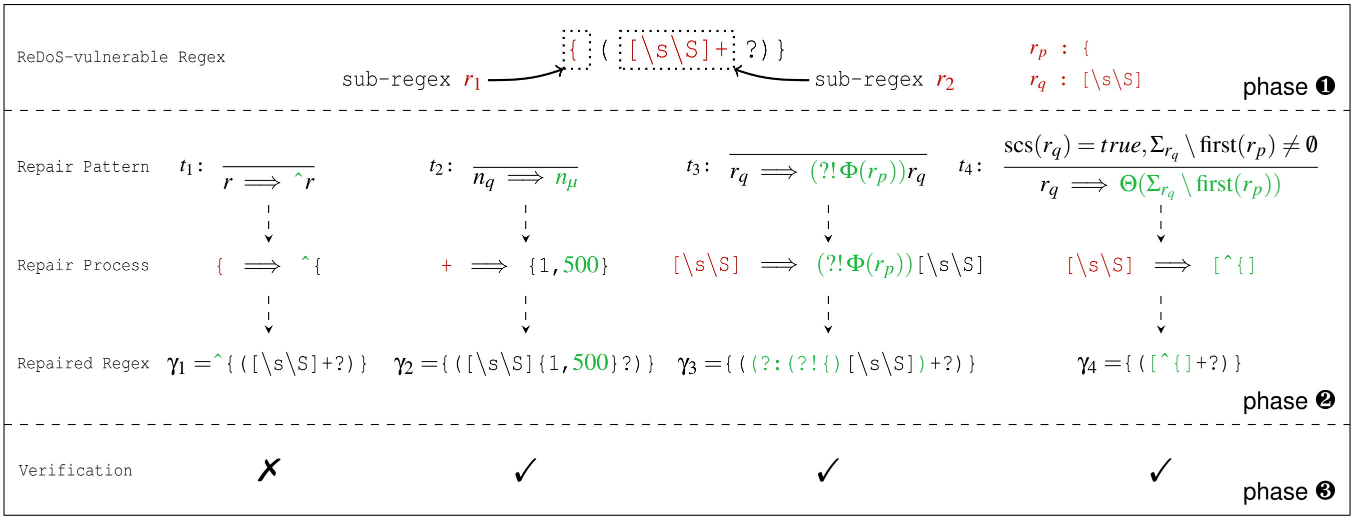

Figure 2: An Example for Repairing the ReDoS-vulnerable Regex `{([\s\S]+?)}`.

by r_p, where $\Phi(r_p)$ returns a regex obtained by transforming all the (named) capturing groups in the regex r_p into the non-capturing ones. By applying the pattern t_3, we can generate a patch γ_3 = `{((?:(?!{)[\s\S])+?)}`[4]. The sub-regex $(?!\Phi(r_p))r_q$ (i.e., `(?!{)[\s\S]`) ensures that r_q (i.e., `[\s\S]`) will not match on `{` that can be matched by r_p (i.e., `{`), that is, the two sub-regexes r_p and $(?!\Phi(r_p))r_q$ will not overlap. So the patch γ_3 can throw away the ReDoS vulnerability. Moreover, we can further focus on the first position of the strings matched by r_p. And if r_q is a character class, we can eliminate this overlap by utilizing a negative set, that is, the fourth pattern t_4 in Figure 2, yielding a patch γ_4 = `{([^{]+?)}` with a character class not overlapping with r_p, where $\Theta(\mathcal{D})$ transforms the set $\mathcal{D}$ into an (equivalent) character class, and $\mathrm{scs}(r_q) = true$ denotes regex r_q is a character class and $\mathrm{scs}(r_q) = false$ otherwise. Indeed, patches γ_3 and γ_4 are equivalent.

The final step of our approach is to determine whether these patches are successful. As demonstrated in the phase ❸ of Figure 2, the patches γ_2, γ_3 and γ_4 are invulnerable to ReDoS attacks and pass the given test cases, so they are successful. While for the patch γ_1, it is easy to verify that γ_1 is invulnerable to ReDoS attacks, but γ_1 fails at the maintainer-provided test case below, because it does not match the non-leading strings formatting using `{}`.

```
exports['basic name substitution'] = function() {
  assert.equal(
    tmpl('the answer is {answer}', { answer: 42 }),
    'the answer is 42')
}
```

We randomly selected the patch γ_4 to report to the maintainers, and the maintainers confirmed it and released nodejs-tmpl 1.0.5 with the fix.

[4] Here, we add a non-capturing group `(?:)` to ensure that the capturing groups of the patch γ_3 and the initial regex are consistent.

4 The Algorithms

4.1 Vulnerability Analysis

Inspired by Li et al. [19] and Liu et al. [24], we diagnose the vulnerability source via localizing the pathological sub-regex, according to the vulnerable patterns.

4.1.1 Nested Quantifiers ($\mathcal{NQ}$)

The first vulnerable pattern is a regex with nested quantifiers, which is called Nested Quantifiers ($\mathcal{NQ}$) pattern [11, 19]. A $\mathcal{NQ}$ pattern has worst-case exponential behavior when a pump string can be consumed by either an inner quantifier or an outer one. In order to facilitate fixing the pathological regex, we subdivide $\mathcal{NQ}$ pattern into three sub-patterns (i.e., $\mathcal{NQ}_1$, $\mathcal{NQ}_2$, and $\mathcal{NQ}_3$), as given in Table 1.

In sub-pattern $\mathcal{NQ}_1$, the entire inner sub-regex is quantified as well. An example of sub-pattern $\mathcal{NQ}_1$ is the key portion `(\d+)*` in the real-world regex δ_1 (Table 1) from CVE-2015-9239. In sub-pattern $\mathcal{NQ}_2$, the inner sub-regex is a form of $r_0r_1r_2$, where r_0 and r_2 are nullable, and r_1 is a quantified regex. An example of sub-pattern $\mathcal{NQ}_2$ is the key portion `([^\s\/?\.#]+\.?)+` in the real-world regex δ_2 (Table 1) from CVE-2021-26272. The sub-pattern $\mathcal{NQ}_3$ is similar to $\mathcal{NQ}_2$, but differs in that r_1 is a disjunction containing a quantified sub-regex as a disjunct. An example of sub-pattern $\mathcal{NQ}_3$ is the key portion `(\[[^\[\]]*\]|\s+)+` in the real-world regex δ_3 (Table 1) from the NPM package moment[5].

[5] https://github.com/moment/moment

Table 1: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaNQ of the Pattern $\mathcal{NQ}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{NQ}_1$	$r = r_1\{m,n\}$, where $r_1 = r_p\{m_p,n_p\}$, $n_p > 1$, and $n > 1$	$\delta_1 =$ `\[(\d+;)?(\d+)*m` (CVE-2015-9239)	($\mathcal{NQ}_1$, `(\d+)*`, [`+`, `*`])
#2	$\mathcal{NQ}_2$	$r = (r_0 r_1 r_2)\{m,n\}$, where r_0 and r_2 are nullable, $r_1 = r_p\{m_p,n_p\}$, $n_p > 1$, and $n > 1$	$\delta_2 =$ `^(https?|ftp):\/\/(-\.)?([^\s\/?\.#]+\.?)+(\/[^\s]*)?[^\s\.,]$` (CVE-2021-26272)	($\mathcal{NQ}_2$, `([^\s\/?\.#]+\.?)+`, [`+`, `+`])
#3	$\mathcal{NQ}_3$	$r = (r_0 r_1 r_2)\{m,n\}$, where r_0 and r_2 are nullable, $r_1 = (\ldots \mid r_p\{m_p,n_p\} \mid \ldots)$, $n_p > 1$, and $n > 1$	$\delta_3 =$ `D[oD]?(\[[^\[\]]*\]|\s+)+MMMM?` (moment)	($\mathcal{NQ}_3$, `(\[[^\[\]]*\]|\s+)+`, [`+`, `+`])

Table 2: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaQOD of the Pattern $\mathcal{QOD}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{QOD}_1$	$r = \alpha\{m,n\}$, $\alpha = (r_1 \mid \ldots \mid r_k)$, and $\exists\ \mathcal{L}(\alpha_1) \cap \mathcal{L}(\alpha_2) \neq \emptyset$, where $\alpha_1 = r_{p_1} r_{p_2} \ldots r_{p_t}$, $\alpha_2 = r_{q_1} r_{q_2} \ldots r_{q_s}$, $1 \leq t,s \leq k$, $1 \leq i \leq t$, $1 \leq j \leq s$, $1 \leq p_i, q_j \leq k$, and $p_1 \neq q_1$	$\delta_4 =$ `^(\d\.|\.\d|\d)+$` ([14])	($\mathcal{QOD}_1$, `(\d\.|\.\d|\d)+`, [`\d\.\d`, `\d\.\d`])
#2	$\mathcal{QOD}_2$	$r = \alpha\{m,n\}$, $\alpha = (r_1 \mid \ldots \mid r_k)$, and $\exists\ \mathcal{L}(r_p) \cap \mathcal{L}(\alpha_1) \neq \emptyset$, $\mathcal{L}(r_l) \cap \mathcal{L}(r_p) = \emptyset$, where $\alpha_1 = r_{q_1} r_{q_2} \ldots r_{q_s}$, $1 \leq i \leq k$, $r_i = r_l r_p$, r_l is not nullable, $1 \leq s \leq k$, $1 \leq j \leq s$, $1 \leq q_j \leq k$	$\delta_5 =$ `(?:[\w-]|\$[-\w]+|#\{\$[-\w]+\})+(?=\s*:)` (CVE-2021-23341)	($\mathcal{QOD}_2$, `(?:[\w-]|\$[-\w]+|#\{\$[-\w]+\})+`, [`[\w-]`, `\$[-\w]+`])

4.1.2 Quantified Overlapping Disjunction ($\mathcal{QOD}$)

The second vulnerable pattern is a quantified disjunction whose multiple inner sub-regexes overlap, which is called Quantified Overlapping Disjunction ($\mathcal{QOD}$) pattern [19]. When matching a pump string, there are multiple choices among the overlapping sub-regex, with worst-case exponential behavior on a mismatch. As shown in Table 2, we subdivide $\mathcal{QOD}$ pattern into two sub-patterns (i.e., $\mathcal{QOD}_1$, $\mathcal{QOD}_2$) for the convenience of fixing the pathological regex.

In sub-pattern $\mathcal{QOD}_1$, a t-iteration (*i.e.*, α_1) of the inner sub-regexes overlaps with another s-iteration (*i.e.*, α_2), starting with different heads (*i.e.*, r_{p_1} and r_{q_1}). An example of sub-pattern $\mathcal{QOD}_1$ is the key portion `(\d\.|\.\d|\d)+` in the regex δ_4 (Table 2) from [14], where $\alpha_1 =$ `\d\.\d` and $\alpha_2 =$ `\d\.\d`. In sub-pattern $\mathcal{QOD}_2$, a strict tail (*i.e.*, r_p) of an inter sub-regex (*i.e.*, r_i) overlaps with an s-iteration (*i.e.*, α_1) starting with r_{q_1} (at least one iteration is not r_i). An example of sub-pattern $\mathcal{QOD}_2$ is the key portion `(?:[\w-]|\$[-\w]+|...)+` in the real-world regex δ_5 (Table 2) from CVE-2021-23341, where $r_p =$ `[-\w]+` and $\alpha_1 =$ `[\w-]`.

4.1.3 Quantified Overlapping Adjacent ($\mathcal{QOA}$)

The third pattern is a quantified regex containing two adjacent overlapping sub-regexes, which is called Quantified Overlapping Adjacent ($\mathcal{QOA}$) pattern [19]. In the pattern $\mathcal{QOA}$, a pump string can be consumed by either of the overlapping adjacencies. To facilitate fixing the pathological regex, we subdivide the pattern $\mathcal{QOA}$ into five sub-patterns, as shown in Table 3. The detailed descriptions and real-world examples are also given in Table 3. Note that, $\mathcal{QOA}_1$ contains two directly adjacent overlapping sub-regexes, while $\mathcal{QOA}_2$, $\mathcal{QOA}_3$, $\mathcal{QOA}_4$, and $\mathcal{QOA}_5$ contain two overlapping sub-regexes that one can reach the other by crossing the quantifier and/or skipping the optional sub-regex. In addition, assuming the outer quantifier is $\{m,n\}$, we also call $\mathcal{QOA}$ as $\mathcal{EOA}$ if $n > 1$ and as $\mathcal{POA}$ otherwise (*i.e.*, $n \leq 1$).

4.1.4 Starting with Large Quantifier ($\mathcal{SLQ}$)

The fourth pattern is a regex starting with a sub-regex with a large quantifier, which is called Starting with Large Quantifier ($\mathcal{SLQ}$) pattern [19]. The $\mathcal{SLQ}$ pattern may cause the regex engine to keep moving the matching regex across the malicious string that does not have a match for the regex, leading to worst-case polynomial behavior.

As shown in Table 4, we subdivide $\mathcal{SLQ}$ pattern into five sub-patterns (i.e., $\mathcal{SLQ}_1$, $\mathcal{SLQ}_2$, $\mathcal{SLQ}_3$, $\mathcal{SLQ}_4$, $\mathcal{SLQ}_5$) for fixing the pathological regex. Sub-pattern $\mathcal{SLQ}_1$ starts with a sub-regex with a large quantifier, and sub-pattern $\mathcal{SLQ}_2$ can behave the same by skipping the beginning nullable sub-regex. While sub-patterns $\mathcal{SLQ}_3$, $\mathcal{SLQ}_4$ and $\mathcal{SLQ}_5$ start with the sub-regex $r_1 r_2$, where r_1 is not nullable, r_2 contains a large outer quantifier, and r_1 overlaps with the prefix, infix and suffix of r_2, respectively. The examples for these five sub-patterns are also given in Table 4.

We propose algorithms AnaNQ, AnaQOD, AnaQOA, and AnaSLQ to check whether the given regex contains any of the $\mathcal{NQ}$, $\mathcal{QOD}$, $\mathcal{QOA}$, and $\mathcal{SLQ}$ sub-patterns. These algorithms will return the vulnerability information as a triple, which consists of a pattern (*i.e.*, the vulnerability sub-pattern), source (*i.e.*, the pathological sub-regex), and cause (*i.e.*, the main components that may cause catastrophic backtracking during matching and would be modified during repairing, such as the overlapping sub-regexes), if any of these sub-patterns was

Table 3: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaQOA of the Pattern $\mathcal{QOA}$.

<table>
<tr><th>No.</th><th>Sub-pattern</th><th>Vulnerability Description</th><th>Example Regex</th><th>Returned Triple</th></tr>
<tr><td>#1</td><td>$\mathcal{QOA}_1$</td><td>$r = (\ldots r_1 r_2 \ldots)\{m,n\}$, where $r_1 = r_p\{m_p,n_p\}$, $r_2 = r_q\{m_q,n_q\}$, $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) \neq \emptyset$</td><td>$\delta_6 =$ <code>^__[^\W_]+\w+__$</code> (pylint)</td><td>($\mathcal{QOA}_1$, <code>[^\W_]+\w+</code>, [<code>[^\W_]+</code>, <code>\w+</code>])</td></tr>
<tr><td>#2</td><td>$\mathcal{QOA}_2$</td><td>$r = (\ldots r_1 r_2 r_3 \ldots)\{m,n\}$, where $r_2 = r_t\{m_t,n_t\}$ is nullable, $r_1 = r_p\{m_p,n_p\}$, $r_3 = r_q\{m_q,n_q\}$, $\mathcal{L}(r_1) \cap \mathcal{L}(r_3) \neq \emptyset$</td><td>$\delta_7 =$ <code>^(>=?|<=?)\s*(\d*\.?\d+)%$</code> (CVE-2021-23364)</td><td>($\mathcal{QOA}_2$, <code>\d*\.?\d+</code>, [<code>\d*</code>, <code>\d+</code>])</td></tr>
<tr><td>#3</td><td>$\mathcal{QOA}_3$</td><td>$r = (r_1 \ldots r_2)\{m,n\}$, where $r_1 = r_p\{m_p,n_p\}$, $r_2 = r_q\{m_q,n_q\}$, $n > 1$, $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) \neq \emptyset$</td><td>$\delta_8 =$ <code>@([\w\-]+\.[\w\-:]+)+[:/]</code></td><td>($\mathcal{QOA}_3$, <code>([\w\-]+\.[\w\-:]+)+</code>, [<code>[\w\-]+</code>, <code>[\w\-:]+</code>])</td></tr>
<tr><td>#4</td><td>$\mathcal{QOA}_4$</td><td>$r = (r_1 \ldots r_2 r_3)\{m,n\}$, where $r_3 = r_t\{m_t,n_t\}$ is nullable, $r_1 = r_p\{m_p,n_p\}$, $r_2 = r_q\{m_q,n_q\}$, $n > 1$, $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) \neq \emptyset$</td><td>$\delta_9 =$ <code>^([ab]+d[ac]+e?)+$</code></td><td>($\mathcal{QOA}_4$, <code>([ab]+d[ac]+e?)+</code>, [<code>[ab]+</code>, <code>[ac]+</code>])</td></tr>
<tr><td>#5</td><td>$\mathcal{QOA}_5$</td><td>$r = (r_1 r_2 \ldots r_3)\{m,n\}$, where $r_1 = r_t\{m_t,n_t\}$ is nullable, $r_2 = r_p\{m_p,n_p\}$, $r_3 = r_q\{m_q,n_q\}$, $n > 1$, $\mathcal{L}(r_2) \cap \mathcal{L}(r_3) \neq \emptyset$</td><td>$\delta_{10} =$ <code>^(;(([\t]*[0-9a-zA-Z]+=[\x21-\x7E]*)*)[\t]*)?$</code></td><td>($\mathcal{QOA}_5$, <code>([\t]*[0-9a-zA-Z]+=[\x21-\x7E]*)*</code>, [<code>[0-9a-zA-Z]+</code>, <code>[\x21-\x7E]*</code>])</td></tr>
</table>

Table 4: The Sub-pattern, Vulnerability Description, Example Regex, and Results from AnaSLQ of the Pattern $\mathcal{SLQ}$.

No.	Sub-pattern	Vulnerability Description	Example Regex	Returned Triple
#1	$\mathcal{SLQ}_1$	Starting with $r = r_1$, where $r_1 = r_q\{m_q,n_q\}$, and $n_q > 1$	$\delta_{11} =$ `([A-Z]+)([A-Z][a-z])` (CVE-2021-3820)	($\mathcal{SLQ}_1$, `[A-Z]+`, `[A-Z]+`)
#2	$\mathcal{SLQ}_2$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is nullable, $r_2 = r_q\{m_q,n_q\}$, and $n_q > 1$	$\delta_{12} =$ `\$?[A-Z]+` (PrismJS)	($\mathcal{SLQ}_2$, `\$?[A-Z]+`, `[A-Z]+`)
#3	$\mathcal{SLQ}_3$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is not nullable, $r_2 = r_q\{m_q,n_q\}$, and $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) \neq \emptyset$	$\delta_{13} =$ `{([\s\S]+?)}` (CVE-2021-3777)	($\mathcal{SLQ}_3$, `{([\s\S]+?)`, [`{`, `[\s\S]+`])
#4	$\mathcal{SLQ}_4$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is not nullable, $r_2 = r_q\{m_q,n_q\}$, $r_q = r_{q_1} r_{q_2}$, r_{q_1} is not nullable, $r_{q_2} = r_t\{m_t,n_t\}$, and $\mathcal{L}(r_{q_2}) \cap \mathcal{L}(r_1) \neq \emptyset$	$\delta_{14} =$ `[ab](ca+)+d`	($\mathcal{SLQ}_4$, `[ab](ca+)+`, [`[ab]`, `a+`])
#5	$\mathcal{SLQ}_5$	Starting with $r = r_1 r_2$, where $r_1 = r_p\{m_p,n_p\}$ is not nullable, $r_2 = r_q\{m_q,n_q\}$, $r_q = r_{q_1} r_{q_2} r_{q_3}$, r_{q_1} and r_{q_3} are not nullable, $r_{q_2} = r_t\{m_t,n_t\}$, and $\mathcal{L}(r_{q_2}) \cap \mathcal{L}(r_1) \neq \emptyset$	$\delta_{15} =$ `[ab](ca{1,2}da)+e`	($\mathcal{SLQ}_5$, `[ab](ca{1,2}da)+`, [`[ab]`, `a{1,2}`])

met.

4.2 Vulnerability Repair

After vulnerability analysis, we can get the vulnerability information (*i.e.*, pattern, source, and cause). In this section, we will introduce how to revise the regex, using our proposed repair patterns targeting at the vulnerability information.

Table 5: The Repair Patterns for Nested Quantifiers ($\mathcal{NQ}$).

No.	Repair Pattern
τ_1	$\dfrac{\mathcal{L}(r) = \mathcal{L}(r_p\{m_p \times m, n_p \times n\})}{r \Longrightarrow (r_p\{m_p \times m, n_p \times n\})\cancel{\{m,n\}}}\ (\mathcal{NQ}_1)$
τ_2	$\dfrac{\mathcal{L}(r) = \mathcal{L}((r_0 r_p \cancel{\{m_p,n_p\}} r_2)\{m,n\})}{r \Longrightarrow (r_0 r_p \cancel{\{m_p,n_p\}} r_2)\{m,n\}}\ (\mathcal{NQ}_2)$
τ_3	$\dfrac{\mathcal{L}(r) = \mathcal{L}((r_0(\ldots \mid r_p \cancel{\{m_p,n_p\}} \mid \ldots)r_2)\{m,n\})}{r_1 \Longrightarrow (\ldots \mid r_p \cancel{\{m_p,n_p\}} \mid \ldots)}\ (\mathcal{NQ}_3)$

4.2.1 Nested Quantifiers ($\mathcal{NQ}$)

After calling AnaNQ, we can get the triple (pattern, source, cause), where pattern belongs to $\mathcal{NQ}_1$, $\mathcal{NQ}_2$, or $\mathcal{NQ}_3$, source is the pathological regex r, and cause contains the nested quantifiers as shown in Table 1. The pathological regex r has, by their nature, a redundant quantifier. So to fix r, we can remove the redundant quantifier.

The repair patterns targeting at $\mathcal{NQ}$ are given in Table 5. Specifically, the repair pattern τ_1 aims to repair the sub-pattern $\mathcal{NQ}_1$ by replacing the inner quantifier $\{m_p,n_p\}$ with the merged quantifier $\{m_p \times m, n_p \times n\}$ and deleting the outer quantifier $\cancel{\{m,n\}}$, provided that the condition $\mathcal{L}(r) = \mathcal{L}(r_p\{m_p \times m, n_p \times n\})$ is satisfied. Let us continue to consider the pathological regex `(\d+)*` from CVE-2015-9239 given in Table 1. Applying the repair pattern τ_1, we can fix it into a *safe* regex `(\d*)`. The repair patterns τ_2 and τ_3 are proposed to repair the sub-patterns $\mathcal{NQ}_2$ and $\mathcal{NQ}_3$, respectively. Both of them remove the inner quantifier $\{m_p,n_p\}$ directly, if the language of the repaired regex is equivalent to the one of the original regex. Taking the pathological regexes `([^\s\/?\.#]+\.?)+` and `(\[[^\[\]]*\]|\s+)+` in Table 1 for example, we can respectively fix them into the safe ones `([^\s\/?\.#]\.?)+` and `(\[[^\[\]]*\]|\s)+` using the repair patterns τ_2 and τ_3.

Table 6: The Repair Patterns for Quantified Overlapping Disjunction ($\mathcal{QOD}$).

No.	Repair Pattern	No.	Repair Pattern	No.	Repair Pattern
τ_4	$\dfrac{\mathcal{L}(r)=\mathcal{L}((r_1\mid\ldots\mid\cancel{r_p}\mid\ldots\mid r_k)\{m,n\})}{\alpha \implies (r_1\mid\ldots\mid\cancel{r_p}\mid\ldots\mid r_k)}\ (\mathcal{QOD}_1)$	τ_5	$\dfrac{t>1}{r_{p_1} \implies r_{p_1}(?!\Phi(r_{p_2}))}\ (\mathcal{QOD}_1)$	τ_6	$\dfrac{s>1}{r_{q_1} \implies r_{q_1}(?!\Phi(r_{q_2}))}\ (\mathcal{QOD}_{\{1,2\}})$
τ_7	$\dfrac{}{r_{p_1} \implies (?!\Phi(\alpha_2))r_{p_1}}\ (\mathcal{QOD}_1)$	τ_8	$\dfrac{}{r_{q_1} \implies (?!\Phi(\alpha_1))r_{q_1}}\ (\mathcal{QOD}_1)$	τ_9	$\dfrac{\mathrm{scs}(\alpha_1)=true,\quad \Sigma_{\alpha_1}\setminus\mathrm{first}(r_{q_1})\neq\emptyset}{\alpha_1 \implies \Theta(\Sigma_{\alpha_1}\setminus\mathrm{first}(r_{q_1}))}\ (\mathcal{QOD}_1)$
τ_{10}	$\dfrac{\mathrm{scs}(\alpha_2)=true,\quad \Sigma_{\alpha_2}\setminus\mathrm{first}(r_{p_1})\neq\emptyset}{\alpha_2 \implies \Theta(\Sigma_{\alpha_2}\setminus\mathrm{first}(r_{p_1}))}\ (\mathcal{QOD}_1)$	τ_{11}	$\dfrac{}{r_p \implies r_p(?<!\Phi(\alpha_1))}\ (\mathcal{QOD}_2)$	τ_{12}	$\dfrac{}{r_{q_1} \implies (?!\Phi(r_p))r_{q_1}}\ (\mathcal{QOD}_2)$
τ_{13}	$\dfrac{\mathrm{scs}(\alpha_1)=true,\quad \Sigma_{\alpha_1}\setminus\mathrm{first}(r_p)\neq\emptyset}{\alpha_1 \implies \Theta(\Sigma_{r_{q_1}}\setminus\mathrm{first}(r_p))}\ (\mathcal{QOD}_2)$	τ_{14}	$\dfrac{r_p=r_u\{m_u,n_u\},\quad \mathrm{scs}(r_u)=true,\quad \Sigma_{r_u}\setminus\mathrm{first}(r_{q_1})\neq\emptyset}{r_p \implies \Theta(\Sigma_{r_u}\setminus\mathrm{first}(r_{q_1}))}\ (\mathcal{QOD}_2)$		

4.2.2 Quantified Overlapping Disjunction ($\mathcal{QOD}$)

The cause returned by AnaQOD is the corresponding two overlapping disjunctions in the pathological regex r (*i.e.*, the source), as illustrated in Table 2. Intuitively, a vulnerable pattern $\mathcal{QOD}$ may have multiple matching paths across the overlapping disjunctions for a string. To repair $\mathcal{QOD}$, we need to ensure that there is a unique matching path for each string. For that, we propose three strategies, namely, deleting one overlapping disjunction, adding a *lookaround* constraint to one overlapping disjunction, and modifying one overlapping disjunction by subtracting the first set of the other one. Table 6 gives our repair patterns for $\mathcal{QOD}$, where $\Phi(r)$ returns a regex obtained by transforming all the (named) capturing groups in the regex r into the non-capturing ones and $\Theta(\mathcal{D})$ transforms the set $\mathcal{D}$ into an (equivalent) *character class*.

To fix the sub-pattern $\mathcal{QOD}_1$, the repair pattern τ_4 removes the overlapping disjunction directly, if it does not alter the language. For example, for the regex `^(ab|a|b)+$`, the overlapping disjunctions (*i.e.*, cause) returned by the algorithm AnaQOD are $\alpha_1 =$ `ab` and $\alpha_2 =$ `ab`. As $\mathcal{L}($`(ab|a|b)+`$) = \mathcal{L}(\texttt{(}\cancel{\texttt{ab}}\texttt{|a|b)+})$, we can use the repair pattern τ_4 to fix `(ab|a|b)+` and get `(a|b)+`. The repair patterns τ_5-τ_8 add a *negative lookaround* (i.e., negative lookahead or negative lookbehind) to specify what does not come before or after a match, and thus ensure that each string has a unique matching path. Consider the above regex `(ab|a|b)+` again. With the repair pattern τ_5, τ_7, or τ_8, we can also fix it into `(ab|a(?!b)|b)+`, `(ab|(?!ab)a|b)+`, or `((?!a)ab|a|b)+`, respectively. The repair patterns τ_9 and τ_{10} handle the case wherein one overlapping disjunction is a *character class*. Specifically, τ_9 and τ_{10} remove the common characters from the *character class* to ensure there is no overlap. For example, we can fix the pathological regex `([ab]|[ac])+` to `^([b]|[ac])+$` and `^([ab]|[c])+$`, according to the repair patterns τ_9 and τ_{10}, respectively. The sub-pattern $\mathcal{QOD}_2$ can be fixed similarly, as shown in Table 6.

4.2.3 Quantified Overlapping Adjacent ($\mathcal{QOA}$)

Similar to $\mathcal{QOD}$, the returned cause for pattern $\mathcal{QOA}$ contains the corresponding two overlapping adjacencies. Likewise, to repair $\mathcal{QOA}$, we need to ensure that only one overlapping adjacency would be selected when matching on a string, and propose three repair strategies, that is, merging the overlapping adjacencies, adding a *lookaround* constraint to one overlapping adjacency, and modifying one overlapping adjacency.

The repair patterns for the sub-pattern $\mathcal{QOA}_1$ are listed in Table 7. First, if the languages of these two overlapping adjacencies are equivalent, the repair pattern τ_{15} simply merges them. For example, for the regex `^a+a+b$`, we can fix it to `^a{2,}b$` using repair pattern τ_{15}. Similar to $\mathcal{QOD}$, the repair patterns τ_{16} and τ_{17} leverage a negative lookaround to resolve the overlap of the adjacencies, thus making the matching path unique for each string. Consider the example regex $\delta_6 =$ `^__[^\W_]+\w+__$` mentioned in Table 3, we can utilize repair pattern τ_{17} to fix δ_6 and get `^__[^\W_]+(?![^\W_])\w+__$`. If a sub-pattern $\mathcal{QOA}_1$ is a $\mathcal{POA}$, (i.e., the outer quantifier $n \leq 1$), the repair pattern τ_{18} substitutes both the quantifiers of these two adjacencies with a small one $n_\mu = 500$, which can cut the number of backtracking and thus avoid ReDoS attacks. For example, for the above regex δ_6, we can also fix it to `^__[^\W_]{1,500}\w{1,500}__$` via the repair pattern τ_{18}. The repair patterns τ_{19} and τ_{20} handle the case wherein one overlapping adjacency is a *character class*. Consider the regex δ_6 again, we leverage repair pattern τ_{20} to fix δ_6 and get `^__[^\W_]+_+__$`.

Likewise, the other sub-patterns (i.e., $\mathcal{QOA}_2$, $\mathcal{QOA}_3$, $\mathcal{QOA}_4$, and $\mathcal{QOA}_5$) can be fixed, and the repair patterns are illustrated in Table 7.

4.2.4 Starting with Large Quantifier ($\mathcal{SLQ}$)

The cause returned by AnaSLQ is either the sub-regex starting with a large quantifier (for $\mathcal{SLQ}_1$ and $\mathcal{SLQ}_2$) or the overlap-

Table 7: The Repair Patterns for Quantified Overlapping Adjacency ($\mathcal{QOA}$).

No.	Repair Pattern	No.	Repair Pattern	No.	Repair Pattern
τ_{15}	$\frac{\mathcal{L}(r_p)=\mathcal{L}(r_q)}{r_1 r_2 \implies r_p\{m_p+m_q, n_p+n_q\}}\ (\mathcal{QOA}_1)$	τ_{16}	$\frac{}{r_1 \implies r_1(?<!\Phi(r_q))}\ (\mathcal{QOA}_1)$	τ_{17}	$\frac{}{r_2 \implies (?!\Phi(r_p))r_2}\ (\mathcal{QOA}_1)$
τ_{18}	$\frac{n \leq 1}{n_p \implies n_\mu, n_q \implies n_\mu}\ (\mathcal{QOA}_{\{1,2\}})$	τ_{19}	$\frac{\mathrm{scs}(r_p)=true,\quad \Sigma_{r_p}\setminus \mathrm{first}(r_q)\neq\emptyset}{r_p \implies \Theta(\Sigma_{r_p}\setminus \mathrm{first}(r_q))}\ (\mathcal{QOA}_{\{1,3\}})$	τ_{20}	$\frac{\mathrm{scs}(r_q)=true,\quad \Sigma_{r_q}\setminus \mathrm{first}(r_p)\neq\emptyset}{r_q \implies \Theta(\Sigma_{r_q}\setminus \mathrm{first}(r_p))}\ (\mathcal{QOA}_{\{1,3\}})$
τ_{21}	$\frac{\mathrm{scs}(r_p)=true,\quad \Sigma_{r_p}\setminus \mathrm{first}(r_q)\neq\emptyset,\quad m_p\geq 1}{r_1 \implies r_p\{m_p-1, n_p-1\}\Theta(\Sigma_{r_p}\setminus \mathrm{first}(r_q))}\ (\mathcal{QOA}_1)$	τ_{22}	$\frac{\mathrm{scs}(r_q)=true,\quad \Sigma_{r_q}\setminus \mathrm{first}(r_p)\neq\emptyset,\quad m_q\geq 1}{r_2 \implies \Theta(\Sigma_{r_q}\setminus \mathrm{first}(r_p))r_q\{m_q-1, n_q-1\}}\ (\mathcal{QOA}_1)$	τ_{23}	$\frac{}{r_1 \implies (?!\Phi(r_q))r_1}\ (\mathcal{QOA}_3)$
τ_{24}	$\frac{}{r_2 \implies r_2(?<!\Phi(r_p))}\ (\mathcal{QOA}_3)$	τ_{25}	$\frac{\mathrm{scs}(r_p)=true,\quad \Sigma_{r_p}\setminus \mathrm{first}(r_q)\neq\emptyset,\quad m_p\geq 1}{r_1 \implies \Theta(\Sigma_{r_p}\setminus \mathrm{first}(r_q))r_p\{m_p-1, n_p-1\}}\ (\mathcal{QOA}_3)$	τ_{26}	$\frac{\mathrm{scs}(r_q)=true,\quad \Sigma_{r_q}\setminus \mathrm{first}(r_p)\neq\emptyset,\quad m_q\geq 1}{r_2 \implies r_q\{m_q-1, n_q-1\}\Theta(\Sigma_{r_q}\setminus \mathrm{first}(r_p))}\ (\mathcal{QOA}_3)$
τ_{27}	$\frac{m_t=0}{r \implies r_1 r_3 \vert r_1 r_t\{1,n_t\}r_3}\ (\mathcal{QOA}_2)$	τ_{28}	$\frac{m_t=0}{r_2 r_3 \implies r_2 \vert r_2 r_t\{1,n_t\}}\ (\mathcal{QOA}_4)$	τ_{29}	$\frac{m_t=0}{r_1 r_2 \implies r_2 \vert r_t\{1,n_t\}r_2}\ (\mathcal{QOA}_5)$

ping sub-regexes (for $\mathcal{SLQ}_3$, $\mathcal{SLQ}_4$, and $\mathcal{SLQ}_5$), as illustrated in Table 4. Intuitively, a vulnerable pattern $\mathcal{SLQ}$ makes the regex engine slide continuously to determine the starting position of the match. To fix $\mathcal{SLQ}$, we need to eliminate or alleviate continuous sliding. For that, we propose four strategies, namely, adding a *start-of-line anchor* ^ to the pathological regex r, replacing the large quantifier in the pathological regex r with a small one, adding a *lookaround* constraint to one overlapping sub-regex, and modifying one overlapping sub-regex by subtracting the first set of the other one. The repair patterns for $\mathcal{SLQ}$ are listed in Table 8. Example 1 mentioned in §3 demonstrates the process of using these repair patterns to repair $\mathcal{SLQ}$.

Finally, we prove our repair patterns preserve the semantics, that is, the language of the repaired regex is a subset of the language of the original one (see Appendix A.1 for the proof).

4.3 Vulnerability Verification

After vulnerability repair, we may get one or more repaired regexes, because there may be more than one repair pattern suitable for the pathological regex. Then for every repaired regex, we check whether it is free from ReDoS attacks (i.e., vulnerability-free) [6] and consistent with the given examples. If so, the repaired regex is called a successful one. While if the repaired regex still suffered from ReDoS attacks, then we will continue the vulnerability analysis and repair it. This is because a ReDoS-vulnerable regex may contain more than one vulnerability. We argue that a vulnerability pattern would not always appear in the repaired regexes again after repair (see Appendix A.2 for the proof). Finally, we will return a repaired regex randomly chosen from the successful ones.

We adopt a simple metric for checking ReDoS that is widely adopted by existing work [11, 12, 26]: a regex is said to be vulnerable to ReDoS attacks if an attack string of fewer than 1 million characters could cause the regex to take 10 seconds or more. In addition, for multiple vulnerabilities in one vulnerable regex, we repair only one vulnerability at a time, and then run the above steps multiple times until there are no vulnerabilities found in the regex.

[6] In this paper, a regex is vulnerability-free means that it is free of four vulnerable patterns (i.e., $\mathcal{NQ}$, $\mathcal{QOD}$, $\mathcal{QOA}$, and $\mathcal{SLQ}$) defined in this paper.

5 Evaluation

In this section, we evaluate RegexScalpel on a wide range of ReDoS-vulnerable regexes collected from the SOLA-DA benchmark [34] and real-world CVEs [9]. We implemented RegexScalpel in Java, and conducted experiments on a machine with 16 cores Intel Xeon CPU E5620 @ 2.40GHz with 12MB Cache, 24GB RAM, running Windows 10. For all the experiments described below, we set the repair time budget for each vulnerability to 5 minutes and the small quantifier n_μ to 500. Our evaluation aims to answer the following three research questions (RQs):

- **RQ1**. **Can RegexScalpel outperform state-of-the-art regex defense techniques?** We evaluate the effectiveness of RegexScalpel on ReDoS-vulnerable regexes from the SOLA-DA benchmark [34] and ReDoS-related CVEs [9], compared with five state-of-the-art ReDoS defense techniques. These techniques vary from regex engine substitution, input length limit and regex repair.
- **RQ2**. **Can RegexScalpel outperform handcrafted defense actions?** We compare the repaired regexes synthesized by RegexScalpel with the defense actions handcrafted by project maintainers.
- **RQ3**. **Can RegexScalpel detect new ReDoS vulnerabilities and synthesize repairs of usefulness to maintainers?** We explore the usefulness of RegexScalpel on popular real-world projects in the detection of new ReDoS vulnerabilities and the synthesis of valid repairs useful to the project maintainers.
- **RQ4**. **Can RegexScalpelsynthesize repaired regexes preserving the semantics of the original ones?** We

Table 8: The Repair Patterns for Starting with Large Quantifier ($\mathcal{SLQ}$).

No.	Repair Pattern	No.	Repair Pattern	No.	Repair Pattern
τ_{30}	$\dfrac{}{r \Longrightarrow \hat{}r}\ (\mathcal{SLQ}_{\{1,2,3,4,5\}})$	τ_{31}	$\dfrac{}{n_q \Longrightarrow n_\mu}\ (\mathcal{SLQ}_{\{1,2,3,4,5\}})$	τ_{32}	$\dfrac{m_p = 0}{r \Longrightarrow r_2 \| r_p\{1, n_p\} r_2}\ (\mathcal{SLQ}_2)$
τ_{33}	$\dfrac{}{r_p \Longrightarrow (?!\Phi(r_q))r_p}\ (\mathcal{SLQ}_3)$	τ_{34}	$\dfrac{}{r_q \Longrightarrow (?!\Phi(r_p))r_q}\ (\mathcal{SLQ}_3)$	τ_{35}	$\dfrac{\mathrm{scs}(r_p) = true, \quad \Sigma_{r_p} \setminus \mathrm{first}(r_q) \neq \emptyset}{r_p \Longrightarrow \Theta(\Sigma_{r_p} \setminus \mathrm{first}(r_q))}\ (\mathcal{SLQ}_3)$
τ_{36}	$\dfrac{\mathrm{scs}(r_q) = true, \quad \Sigma_{r_q} \setminus \mathrm{first}(r_p) \neq \emptyset}{r_q \Longrightarrow \Theta(\Sigma_{r_q} \setminus \mathrm{first}(r_p))}\ (\mathcal{SLQ}_3)$	τ_{37}	$\dfrac{}{r_p \Longrightarrow r_p(?<!\Phi(r_t))}\ (\mathcal{SLQ}_{\{4,5\}})$	τ_{48}	$\dfrac{}{r_t \Longrightarrow (?!\Phi(r_p))r_t}\ (\mathcal{SLQ}_{\{4,5\}})$
τ_{39}	$\dfrac{\mathrm{scs}(r_p) = true, \quad \Sigma_{r_p} \setminus \mathrm{first}(r_t) \neq \emptyset}{r_p \Longrightarrow \Theta(\Sigma_{r_p} \setminus \mathrm{first}(r_t))}\ (\mathcal{SLQ}_{\{4,5\}})$	τ_{40}	$\dfrac{\mathrm{scs}(r_t) = true, \quad \Sigma_{r_t} \setminus \mathrm{first}(r_p) \neq \emptyset}{r_t \Longrightarrow \Theta(\Sigma_{r_t} \setminus \mathrm{first}(r_p))}\ (\mathcal{SLQ}_{\{4,5\}})$		

Table 9: The ReDoS-vulnerable Regex Sets for Evaluation.

Benchmark	#Regex	Description
SOLA-DA [34]	34	ReDoS-vulnerable regexes in NPM modules found by the Software Lab at TU Darmstadt
CVE [9]	414	ReDoS-vulnerable regexes extracted from 70 ReDoS-related CVEs in recent three years
Total	448	

verify that the repaired regexes are subsets of the original ones, and analyze the semantic similarities between the repaired regexes and the original ones.

5.1 Evaluation Setup

5.1.1 Evaluation Datasets

Our evaluation was conducted on the ReDoS-vulnerable regexes collected from two widely-used sources: (i) the SOLA-DA benchmark [34] and (ii) real-world CVEs [9]. The statistics are listed in Table 9. The first benchmark SOLA-DA was constructed by Staicu and Pradel [34]. It consists of 34 ReDoS-vulnerable regexes, and is often used for research on finding, fixing, and mitigating ReDoS vulnerabilities. For the second benchmark, we collected ReDoS-vulnerabilities in the last three years from widely-used libraries with Common Vulnerabilities and Exposures (CVE) [9] identifiers. Collected vulnerabilities without clear descriptions, live links, or test cases were discarded, resulting in 70 CVEs in total. Moreover, one CVE may contain more than one vulnerable regex. Finally, we extracted 414 ReDoS-vulnerable regexes from these 70 CVEs. In total, there are 448 (34 + 414) ReDoS-vulnerable regexes in our evaluation dataset. The details of these regexes together with their test cases can be found online [21].

5.1.2 Evaluation Approaches

To answer RQ1, we selected three state-of-the-art tools belonging to three paradigms, i.e., regex engine substitution (RE2 [16]), input length restriction (LLI[7]), and regex repair (FlashRegex [22]). Since RE2 and LLI do not defend ReDoS vulnerabilities by regex repair, we use the term "defense" instead of "repair" when referring to the evaluation results in RQ1. To answer RQ2, we compared RegexScalpel with the project maintainers.

5.1.3 Evaluation Metrics

A defense is considered successful if it (i) passes all the given test cases, and (ii) is free from ReDoS attack. The *success defense rate* is calculated by dividing the number of successful defenses by the total number of vulnerable regexes under defense.

5.2 RQ1: Comparing State-of-the-art

Overall Results. We evaluated the effectiveness of RegexScalpel on 448 ReDoS-vulnerable regexes comparing with different defense tools, i.e., RE2, LLI with different input length limits, and FlashRegex. Table 10[8] shows the number (ratio) of regexes that have been defended successfully. We can see that across the two benchmarks, the effectiveness of RegexScalpel outperforms existing works, defending over 98% vulnerable regexes, over four times of the best results (FlashRegex with 21.20%) achieved by baselines.

In the following, we further investigated the reason why existing works performed unsatisfactorily, and thus revealed the advantage of RegexScalpel from two perspectives.

Defense Capabilities on Vulnerable Pattern $\mathcal{SLQ}$. We analyzed the distribution of four vulnerable patterns over all

[7]We implemented three variants of input length restriction to limit the length of the input to 100, 500, and 5000, which are denoted as LLI(100), LLI(500), and LLI(5000)), respectively.

[8]Table 10 shows overall results. The detailed results can be found online [21].

Table 10: Success Defense Rate Across Automated Tools.

Tool	SOLA-DA	CVE	Total
RE2	18 (52.94%)	35 (8.45%)	53 (11.83%)
LLI(100)	22 (64.71%)	45 (10.87%)	67 (14.96%)
LLI(500)	26 (76.47%)	18 (4.35%)	44 (9.82%)
LLI(5000)	26 (76.47%)	19 (4.59%)	45 (10.04%)
FlashRegex	4 (11.76%)	91 (21.98%)	95 (21.20%)
RegexScalpel	**33 (97.06%)**	**410 (99.03%)**	**443 (98.88%)**
#Regex	34	414	448

Table 11: Success Defense Rate for $\mathcal{SLQ}$ Regexes.

Tool	SOLA-DA	CVE	Total
RE2	16 (55.17%)	21 (6.75%)	37 (10.88%)
LLI(100)	21 (72.41%)	26 (8.36%)	47 (13.82%)
LLI(500)	26 (89.66%)	18 (5.79%)	44 (12.94%)
LLI(5000)	26 (89.66%)	18 (5.79%)	44 (12.94%)
FlashRegex	0 (0%)	0 (0%)	0 (0%)
RegexScalpel	**29 (100%)**	**309 (99.36%)**	**338 (99.41%)**
#Regex	29	311	340

vulnerable regexes, and observed that the pattern $\mathcal{SLQ}$ is the major cause of the vulnerability, accounting for 85.29% (29 / 34) and 75.12% (311 / 414) ReDoS vulnerabilities in the two benchmarks, respectively. We further analyzed the results of the tools for their defending vulnerabilities with respect to $\mathcal{SLQ}$ pattern. The statistics are shown in Table 11. We note that the replacement of regex engines (i.e., RE2) only reaches 10.88% success defense rate, which indicates that it cannot effectively address vulnerable regexes with $\mathcal{SLQ}$ pattern in both benchmarks. By limiting input length (i.e., LLI), the success defense rate ranges from 5.79% to 89.66%, which is still not satisfactory. In addition, FlashRegex, the state-of-the-art regex defense tool, cannot defend regexes with $\mathcal{SLQ}$ vulnerable pattern, resulting in 0% success defense rate in both benchmarks. In contrast, RegexScalpel can successfully defend 99.41% (338 over 340) regexes with $\mathcal{SLQ}$ pattern, making them free from ReDoS attacks.

Defense Capabilities on Lookarounds and Backreferences. The versatility of the rich extended features is also a reason that sets existing tools back. Among the extended features, we observed two features (i.e., lookarounds and backreferences) are more dominant than other features. So we investigated the effectiveness of regexes with these two features, as shown in Table 12. There are 7 and 32 regexes using lookarounds or backreferences in the two benchmarks, respectively. The design of RE2 and FlashRegex do not consider lookarounds and backreferences, leaving them incapable of defending such vulnerable regexes. The state-of-the-art defense tools (i.e., LLI(500) and LLI(5000)) can only defend

Table 12: Success Defense Rate for Regexes with Lookarounds or Backreferences.

Tool	SOLA-DA	CVE	Total
RE2	0 (0%)	0 (0%)	0 (0%)
LLI(100)	0 (0%)	2 (6.25%)	2 (5.13%)
LLI(500)	5 (71.43%)	2 (6.25%)	7 (17.95%)
LLI(5000)	5 (71.43%)	2 (6.25%)	7 (17.95%)
FlashRegex	0 (0%)	0 (0%)	0 (0%)
RegexScalpel	**6 (85.71%)**	**30 (93.75%)**	**36 (92.31%)**
#Regex	7	32	39

at most 7 (17.95%) of them, while our RegexScalpel can successfully defend 92.31% (36 over 39). The table shows the existing tools rarely consider handling vulnerable regexes using lookarounds or backreferences, resulting in the unsatisfactory defense capabilities for these sorts of vulnerabilities.

Finally, we explain the reason why there are five (5/448, 1.1%) regexes which cannot be fixed by RegexScalpel. A fix is considered to be *successful* if: (i) no vulnerabilities can be found, and (ii) all the test cases pass. In our evaluation, the unsuccessful fixes are all caused by the presence of failing test cases. That is, some test cases would not be accepted/matched by the repaired regex.

Summary to RQ1: RegexScalpel can effectively defend 98.88% of vulnerable regexes, compared with 21.20% achieved by the best work. The high defense capability of RegexScalpel benefits from the comprehensive consideration of all four vulnerable patterns. In addition, the design of RegexScalpelconsiders the extended features such as lookarounds and backreferences, enabling RegexScalpel to be applicable for regexes with such extended features.

5.3 RQ2: Comparing Maintainers' Repairs

Overall Results. Apart from the automatic defense tools shown above, we compared the repairs made by RegexScalpel with the defending actions taken by the project maintainers. Defending actions can be regex engine substitution, input length limit, regex repair, or code logic modification. We showed the statistics of maintainers' defense strategies in Table 13. Among four strategies, *regex repair* is mostly used by maintainers, accounting for the highest proportion, as high as 92.19%, followed by code logic modification (5.13%). Also note that there are 8 outstanding vulnerable regexes (1.79%) whose defense actions have not been determined. To answer RQ2, we evaluated RegexScalpel on the 413 vulnerable regexes that were fixed by the maintainers using regex repair.

Table 13: The proportion of Maintainers' Defense Actions.

Defense Strategy	SOLA-DA	CVE	Total
Regex Engine Substitution	0 (0%)	1 (0.24%)	1 (0.22%)
Input Length Limit	1 (2.94%)	2 (0.48%)	3 (0.67%)
Code Logic Modification	6 (17.65%)	17 (4.11%)	23 (5.13%)
Regex Repair	**21 (61.76%)**	**392 (94.69%)**	**413 (92.19%)**
No Fix	6 (17.65%)	2 (0.48%)	8 (1.79%)
#Regex	34	414	448

Table 14: Success Defense Rate of the Repairs by Maintainers and RegexScalpel.

	SOLA-DA	CVE	Total
Manual Repair	14 (66.67%)	305 (77.81%)	319 (77.23%)
RegexScalpel	**21 (100%)**	**388 (98.98%)**	**409 (99.03%)**
#Regex	21	392	413

Table 14 shows that the repairs handcrafted by project maintainers can successfully defend 77.23% (319 / 413) vulnerable regexes as compared with 99.03% (409 / 413) by the ones generated using RegexScalpel. In the following, we discuss our observations from two perspectives, multiple vulnerabilities in one regex and $\mathcal{SLQ}$-pattern regexes.

Multiple Vulnerabilities in One Regex. On examining the vulnerable regexes in the benchmarks, we observed there are vulnerable regexes that contain more than one vulnerability. We conducted further analysis on the number of such regexes that can be successfully repaired by the maintainers and by RegexScalpel, respectively. As shown in Table 15, there are 7 and 191 vulnerable regexes containing more than one vulnerability in the two benchmarks. Maintainers can only repair 57.58% of such regexes, with 14.29% and 59.16% success rates for the benchmarks, respectively. In comparison, RegexScalpel can successfully repair 97.98% of these vulnerable regexes automatically.

ReDoS-vulnerable Regexes with Pattern $\mathcal{SLQ}$. Among the 413 vulnerable regexes that were repaired by maintainers, we made a comparison over the ones with $\mathcal{SLQ}$ pattern. We found 318 such vulnerable regexes from both benchmarks. Table 16 shows the comparison results. Maintainers

Table 15: Success Defense Rate for Multiple Vulnerabilities Regexes of the Repairs by Maintainers and RegexScalpel.

	SOLA-DA	CVE	Total
Manual Repair	1 (14.29%)	113 (59.16%)	114 (57.58%)
RegexScalpel	**7 (100%)**	**187 (97.91%)**	**194 (97.98%)**
#Regex	7	191	198

Table 16: Success Defense Rate for $\mathcal{SLQ}$ Regexes of the Repairs by Maintainers and RegexScalpel.

	SOLA-DA	CVE	Total
Manual Repair	13 (72.22%)	229 (76.33%)	242 (76.10%)
RegexScalpel	**18 (100%)**	**298 (99.33%)**	**316 (99.37%)**
#Regex	18	300	318

can successfully repair 72.22% and 76.33% of the regexes with $\mathcal{SLQ}$ in the two benchmarks, respectively. In comparison, RegexScalpel can achieve 100% and 99.33% in the two benchmarks, respectively. The results reveal that fixing regexes with $\mathcal{SLQ}$ pattern manually is challenging.

> **Summary to RQ2:** Among the 413 repaired vulnerable regexes handcrafted by the maintainers, only 319 (77.23%) are ReDoS free. In RegexScalpel outperforms manual fixing, successfully repairs 409 (99.03%) of the 413 regexes. In particular, RegexScalpel excels at repairing regexes with multiple vulnerabilities and regexes with $\mathcal{SLQ}$ vulnerable pattern.

5.4 RQ3: Usefulness to Maintainers

We explored whether RegexScalpel can detect new ReDoS vulnerabilities in popular real-world projects and synthesize repairs of usefulness to project maintainers. We applied RegexScalpel to ten popular projects including Python and NLTK. The results are shown in Table 17. Although our methodology in this paper focuses on regex repair, the modules in RegexScalpel can also be applied to detecting ReDoS vulnerabilities. In total, RegexScalpel reported 16 new ReDoS regexes from ten projects. We applied RegexScalpel to repair these vulnerable regexes, and reported the repairs to the concerned project maintainers. All the 16 repairs were accepted by the maintainers and merged into subsequent project releases. At the time of submission, 8 CVEs concerning 8 projects and 13 of the 16 vulnerable regexes were assigned. The results demonstrate the usefulness of RegexScalpel to project maintainers in defending against ReDoS attacks.

We take case #1 (i.e., the Python project) in Table 17 as an example to demonstrate the process of applying RegexScalpel in practice. The case concerns a vulnerable regex `(?:.*,)*[ \t]*([^ \t]+)[ \t]+realm=(["\']?)([^"\']*)\2` used by Python built-in module *urllib*. The regex contains six vulnerabilities. Processing of the regex can be computationally expensive and is therefore vulnerable to ReDoS attacks. The Python maintainers replaced the pathological sub-regex `(?:.*,)*` with the safe one `(?:^|,)` to get the patch

Table 17: Demographics of New Vulnerabilities Repaired by RegexScalpel.

No.	Project	Disclosure Date	CVE ID	#Vuln. Regex
#1	Python	Jan 30th, 2021	CVE-2021-3733	1
#2	NLTK	Sep 5th, 2020	–	1
#3	pylint	Sep 3rd, 2020	–	2
#4	mpmath	Oct 8th, 2021	CVE-2021-29063	1
#5	browserslist	Apr 28th, 2021	CVE-2021-23364	6
#6	code-server	Sep 17th, 2021	CVE-2021-3810	1
#7	ansi-regex	Sep 12th, 2021	CVE-2021-3807	1
#8	nth-check	Sep 17th, 2021	CVE-2021-3803	1
#9	nodejs-tmpl	Sep 15, 2021	CVE-2021-3777	1
#10	jspdf	Feb 12th, 2021	CVE-2021-23353	1
			Total	16

`(?:^|,)[ \t]*([^ \t]+)[ \t]+realm=(["\']?)([^"\' ]*)\2`. The patch fixed the first four vulnerabilities related to the pathological sub-regex `(?:.*,)*` in Table 18, and prevented the pathological sub-regex `[ \t]*` from being used as a starting sub-regex, which in turn fixed the fifth vulnerability. However, it cannot fix the sixth vulnerability, which is related to the pathological sub-regex `[^ \t]+`. RegexScalpel analyzed the vulnerability information of the patch ($\mathcal{SLQ}_3$, `(?:^|,)[ \t]*([^ \t]+)`, [`,`, `([^ \t]+)`]). It leveraged the pattern τ_{36} in Table 8 to synthesize a repaired regex `(?:^|,)[ \t]*([^ \t,]+)[ \t]+realm=(["\']?)([^"\' ]*)\2` based on the patch provided by the maintainers. In addition, RegexScalpel leveraged the repair patterns τ_{19} in Table 7 and τ_{30} in Table 8 to repair the initial regex and synthesize another patch `^(?:[^,]*,)*[ \t]*([^ \t]+)[ \t]+realm=(["\']?)([^"\']*)\2`. The two synthesized patches RegexScalpel passed all test cases, and the first one is committed to the maintainers. Finally, the maintainers confirmed and released it in Python 3.6-3.10, and appreciated our help.

Table 18: The Six ReDoS Vulnerabilities in Python built-in module *urllib*.

No.	Sub-pattern	Vuln. Source	Vuln. Cause
❶	$\mathcal{QOA}_1$	`(?:.*,)*`	[`.*`, `,`]
❷	$\mathcal{QOA}_2$	`(?:.*,)*[ \t]*([^ \t]+)`	[`(?:.*,)*`, `([^ \t]+)`]
❸	$\mathcal{SLQ}_1$	`(?:.*,)*`	`(?:.*,)*`
❹	$\mathcal{SLQ}_1$	`.*`	`.*`
❺	$\mathcal{SLQ}_2$	`(?:.*,)*[ \t]*`	`[ \t]*`
❻	$\mathcal{SLQ}_2$	`(?:.*,)*[ \t]*([^ \t]+)`	`[^ \t]+`

Summary to RQ3: RegexScalpel is useful to synthesize non-trivial and effective repairs for new vulnerable regexes found in popular real-world projects.

5.5 RQ4: Semantics Preservation

Finally, we showed the empirical evidence by verifying that the repaired regexes synthesized by RegexScalpel are subsets of the original ones. The experiments indicated that all repaired regexes synthesized by RegexScalpel are subsets of the corresponding original ones.

Furthermore, we calculated the semantic similarities between the repaired regexes and the original ones. To measure the semantic similarity, we used the equation as follows:

$$Sim(r_1, r_2) = \frac{|\mathcal{L}(r_1) \cap \mathcal{L}(r_2)|}{|\mathcal{L}(r_1) \cup \mathcal{L}(r_2)|} \tag{1}$$

where $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$ are languages of regexes r_1 and r_2, respectively. Yet, the languages can be infinite, so we adopted the following equation used in previous work [5]:

$$Sim(r_1, r_2) = \lim_{\lambda \to +\infty} \frac{|\mathcal{L}(r_1)^{\leq\lambda} \cap \mathcal{L}(r_2)^{\leq\lambda}|}{|\mathcal{L}(r_1)^{\leq\lambda} \cup \mathcal{L}(r_2)^{\leq\lambda}|} \tag{2}$$

where $\lambda \in \mathbb{N}$. Formally, for a language $\mathcal{L}(r)$, let $|\mathcal{L}(r)^{\leq\lambda}|$ denote the number of words in $\mathcal{L}(r)$ of length at most λ. The calculation proceeds iteratively until the solution converges to 0.001 as set in [5].

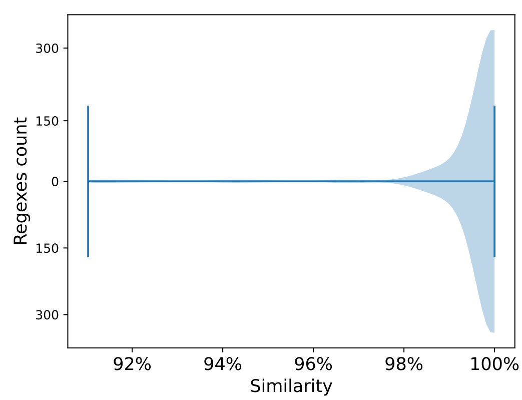

Figure 3: Semantic similarities between the vulnerable regexes and the repaired ones.

Figure 3 visualizes the semantic similarity between the vulnerable regex and the repaired one. We can see that most similarities go beyond 98%. On average, the semantic similarity is 99.57%, meaning that the semantics of regexes are well-preserved after the repair.

Summary to RQ4: RegexScalpel can synthesize repaired regexes preserving the semantics of the original ones and keeping the semantics as close as possible to the original ones.

6 Related Work

ReDoS Defense. Various techniques [2, 7, 8, 10, 13, 15–17, 22, 23, 27–30, 37–39, 42] proposed to defense or mitigate ReDoS vulnerabilities by modifying the structure of regexes (i.e., regex repair) or optimizing regex matching (i.e., regex engine optimization).

Regex Repair. Li et al. [22] proposed the first programming-by-example (PBE) framework, using determinism to integrate regex synthesis and repair with respect to ReDoS vulnerabilities. But it can not synthesize and repair regexes with extended features, such as lookarounds and backreferences. Chida and Terauchi [7] proposed another PBE repair ReDoS framework, which uses strong determinism constraints and supports some extended features (e.g., lookarounds and backreferences). But strong determinism constraints are not necessary. For example, the regex `a(b*|b*)c` is not strongly deterministic but invulnerable. Also, as discussed in Sec. 1, there exist several common problems in [22] and [7]. First, the repair quality depends on the quality of the examples provided by the users. It is difficult for users to provide sufficient (characteristic) examples [20, 25], which makes it difficult for these tools to get a repaired regex that is semantically equivalent or similar to the original one. Second, these tools ignore the $\mathcal{SLQ}$ vulnerable pattern, which leads to the incapability of repairing $\mathcal{SLQ}$ regexes. For example, they can not repair the regex `a+b`. Van der Merwe et al. [39] proposed several DFA-based optimizing techniques transforming vulnerable pure regular expressions into safe ones while focusing only on exponential vulnerabilities, thus this work does not support extended functionalities and is unable to fix polynomial vulnerabilities (e.g., $\mathcal{SLQ}$ and $\mathcal{POA}$ patterns). Cody-Kenny et al. [8] presented a genetic-programming based tool to get alternative regexes with better running time performance, which may synthesize ReDoS-vulnerable regexes and does not support extended features.

Regex Engine Optimization. For eliminating or alleviating ReDoS, many works are dedicated to optimizing the regex engine, e.g., by parallel algorithms [23], GPU-based algorithms [42], state-merging algorithms [2], Thompson's Non-deterministic Finite Automaton algorithm [10, 37], counting automata matching algorithm [38], Parsing Expression Grammars (PEGs) [15, 17, 27], memoization-based optimization [13] and recursion-limit/backtracking-limit/time-limit [28–30]. Whereas significantly improving performance and thus alleviating ReDoS attacks, these techniques also have some obvious flaws, such as not supporting extended features or sacrificing memory.

ReDoS Detection. Previous works [19, 24, 26, 31–33, 35, 36, 40, 41] have studied ReDoS detection, which can be mainly classified as static analysis [31, 32, 40, 41], dynamic analysis [26, 33, 35, 36], and hybrid approaches combining both static and dynamic analysis [19, 24]. It is worth mentioning that there are two detectors (i.e., [19] and [24]) to detect ReDoS vulnerabilities by formally and comprehensively modeling vulnerable patterns. In particular, pattern $\mathcal{SLQ}$ proposed by [19] is a new ReDoS pattern that has not been recognized by previous detection work. Although the vulnerable patterns proposed by [19] and [24] are comprehensive, they are not fine-grained enough for us to directly apply to Sec. 4.1. We took inspiration from Li et al. [19] and Liu et al. [24] and presented fine-grained vulnerable sub-patterns based on [19].

7 Conclusion

In this paper, we propose RegexScalpel, which can defend ReDoS attacks by automatically localizing and repairing ReDoS-vulnerable regexes. RegexScalpel is the first approach to localize and fix the vulnerable regexes considering comprehensive and fine-grained types of vulnerable causes. The experiment reveals the high effectiveness of RegexScalpel by successfully fixing 98.88% vulnerable regexes, compared with 21.20% achieved by the best existing work. Further, it outperformed the manual repair, achieving 99.03% success repair ratio compared with only 77.23% for manual repair. It also helped to repair 16 ReDoS vulnerabilities in the ten real-world projects and got confirmed by the maintainers, resulting in 8 confirmed CVEs.

Acknowledgments

The authors would like to thank the reviewers and our shepherd, Engin Kirda, for their helpful feedback on an earlier version of this paper. This work was supported in part by National Natural Science Foundation of China (Grants #61872339, #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, Shenzhen Fundamental Research Program (Grants #20200810150421002, #JCYJ20200813091134001), the Hong Kong RGC/GRF (Grant No. 16207120), Hong Kong ITF (Grant No. MHP/055/19), Huawei PhD Fellowship, MSRA Collaborative Research Grant, the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRFI06-2020-0022-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award No. NRF2018NCR-NSOE003-0001 and the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

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] 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/.

[4] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A Formal Study Of Practical Regular Expressions. *Int. J. Found. Comput. Sci.*, 14(6):1007–1018, 2003.

[5] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. SemMT: A Semantic-based Testing Approach for Machine Translation Systems. *ACM Trans. Softw. Eng. Methodol.*, 31(2):34e:1–34e:36, 2022.

[6] 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.

[7] Nariyoshi Chida and Tachio Terauchi. Repairing dos vulnerability of real-world regexes. *CoRR*, abs/2010.12450, 2020.

[8] 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.

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

[10] Russ Cox. Regular Expression Matching Can Be Simple And Fast (But Is Slow In Java, Perl, PHP, Python, Ruby, ...), 2007. https://swtch.com/~rsc/regexp/regexp1.html.

[11] 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.

[12] 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.

[13] James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (redos). In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1–17. IEEE, 2021.

[14] James Collins Davis. *On the Impact and Defeat of Regular Expression Denial of Service*. PhD thesis, Virginia Tech, 2020.

[15] 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.

[16] Google. RE2, 2020. https://github.com/google/re2.

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

[18] 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.

[19] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3847–3864. USENIX Association, 2021.

[20] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. TRANSREGEX: Multi-modal Regular Expression Synthesis by Generate-and-Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1210–1222. IEEE, 2021.

[21] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. RegexScalpel, 2022. https://sites.google.com/view/regexscalpel/.

[22] 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.

[23] 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.

[24] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1468–1484. IEEE, 2021.

[25] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating Programming by Example and Natural Language Programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, 2013.

[26] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. Regulator: Dynamic Analysis to Detect ReDoS. In *31th USENIX Security Symposium, USENIX Security 2022, August 10–12, 2022*. USENIX Association, 2022.

[27] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From regexes to parsing expression grammars. *Sci. Comput. Program.*, 93:3–18, 2014.

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

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

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

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

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

[33] 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.

[34] 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.

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

[36] 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.

[37] Ken Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[38] Lenka Turonová, 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.

[39] 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.

[40] 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.

[41] 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.

[42] Xiaodong Yu and Michela Becchi. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 18:1–18:10. ACM, 2013.

A Appendix

A.1 Proof of Semantics Preservation After Repair

Lemma 1. *Given two regexes $r_1 = \alpha\gamma\beta$ and $r_2 = \alpha\delta\beta$, if $\mathcal{L}(\gamma) \subseteq \mathcal{L}(\delta)$, then $\mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$.*

Proof. Since $\mathcal{L}(\gamma) \subseteq \mathcal{L}(\delta)$, $\mathcal{L}(r_1) = \mathcal{L}(\alpha)\mathcal{L}(\gamma)\mathcal{L}(\beta)$ and $\mathcal{L}(r_2) = \mathcal{L}(\alpha)\mathcal{L}(\delta)\mathcal{L}(\beta)$, then we have $\mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$. □

Theorem 1. *Given a ReDoS-vulnerable regex r, RegexScalpel returns the repaired regex r' satisfying that $\mathcal{L}(r') \subseteq \mathcal{L}(r)$.*

Proof. For repairing vulnerable regexes, RegexScalpel uses three kinds of strategies (i.e., adding a lookaround/start-of-line anchor, deleting a quantifier/sub-regex and modifying a quantifier/sub-regex).

According to the repair patterns we proposed, $\tau_1 - \tau_3$ are based on deleting a quantifier. We can see that these patterns can only be used when the two languages before and after deleting the quantifier are equivalent. According to Lemma 1, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$. τ_4 is based on deleting a sub-regex, which can only be used when the two languages before and after deleting are equivalent. Similarly, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$. τ_{18} and τ_{31} are based on modifying quantifiers. For a sub-regex with a large quantifier, if we replace the quantifier with a smaller one (i.e., 500), it's obvious that the language of the sub-regex replaced with a small quantifier is the subset of the language of the sub-regex with a large quantifier. Therefore, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ according to Lemma 1. $\tau_9 - \tau_{10}$, $\tau_{13} - \tau_{15}$, $\tau_{19} - \tau_{22}$, $\tau_{25} - \tau_{29}$, τ_{32}, $\tau_{35} - \tau_{36}$, $\tau_{39} - \tau_{40}$ are based on modifying sub-regex. Among these repair patterns, τ_{15} can only be used when the two languages before and after modifying are equivalent. $\tau_{27} - \tau_{29}$, τ_{32} rewrite the regex by removing its nullable sub-regex, meaning that the regexes after modifying are subsets of the original regexes. Other patterns given above are based on modifying a sub-regex by removing some elements from a character class. Accordingly, for all of these patterns, $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ can be proved based on Lemma 1. $\tau_5 - \tau_8$, $\tau_{11} - \tau_{12}$, $\tau_{16} - \tau_{17}$, $\tau_{23} - \tau_{24}$, $\tau_{33} - \tau_{34}$, $\tau_{37} - \tau_{38}$ are based on adding a lookaround. According to the semantics of lookaround, the language of the sub-regex after adding a lookaround is a subset of the original ones. Therefore, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ according to Lemma 1. τ_{30} is based on adding a start-of-line anchor ^. Because ^ is equivalent to lookaround `(?!.*)`, it is obvious that the language of the sub-regex adding a start-of-line anchor is a subset of the language of the original sub-regex. Therefore, we can prove $\mathcal{L}(r') \subseteq \mathcal{L}(r)$ too. □

A.2 Proof of Vulnerability-free After Repair

Lemma 2. *Given a ReDoS-vulnerable regex $r = \alpha\gamma\beta$ where γ is the pathological sub-regex, RegexScalpel returns the repaired regex $r' = \alpha\delta\beta$ where δ is a safe regex obtained by repairing γ with our repair patterns and $\mathcal{L}(\delta) \subseteq \mathcal{L}(\gamma)$. If δ contains a construct overlapping with another construct in sub-regex α or β, then γ contains this type of construct too, which overlaps with the same construct in α or β.*

Proof. For repairing vulnerable regexes, RegexScalpel uses three kinds of strategies (i.e., adding a lookaround/start-of-line anchor, deleting a quantifier/sub-regex and modifying a quantifier/sub-regex).

According to the repair patterns we proposed, $\tau_1 - \tau_3$ are based on deleting a nested quantifier, and τ_4 is based on deleting a sub-regex (i.e., an inner sub-regex in a disjunction). Thus, deleting a quantifier/sub-regex ensures that γ contains the same type of construct as δ which overlaps with another construct in sub-regex α or β, and the construct in γ overlaps with the same construct in α or β too. τ_{18} and τ_{31} are based on modifying quantifiers which replace a large quantifier with a smaller one (i.e., 500), it is obvious that γ contains the same type of construct as δ which overlaps with another construct in sub-regex α or β, and the construct in γ overlaps with the same construct in α or β too. $\tau_9 - \tau_{10}$, $\tau_{13} - \tau_{15}$, $\tau_{19} - \tau_{22}$, $\tau_{25} - \tau_{29}$, τ_{32}, $\tau_{35} - \tau_{36}$, $\tau_{39} - \tau_{40}$ are based on modifying sub-regex. Among these repair patterns, τ_{15} merges the directly adjacent regexes with a quantified sub-regex (i.e. r_p illustrated in τ_{15}). $\tau_{27} - \tau_{29}$, τ_{32} rewrite the regex by removing its nullable sub-regex. Other patterns given above are based on modifying a sub-regex by removing some elements from a character class which will not change the type of regex construct. Accordingly, for all of these patterns based on modifying sub-regex, we can conclude that if δ contains a construct overlapping with another construct in sub-regex α or β, then γ contains this type of construct too, which overlaps with the same construct in α or β. $\tau_5 - \tau_8$, $\tau_{11} - \tau_{12}$, $\tau_{16} - \tau_{17}$, $\tau_{23} - \tau_{24}$, $\tau_{33} - \tau_{34}$, $\tau_{37} - \tau_{38}$ are based on adding a lookaround. Because lookarounds are zero-length assertions which don't consume characters, adding a lookaround according to the above repair patterns will not change the regex construct. τ_{30} is based on adding a start-of-line anchor `^`. As the proof of Theorem 1, `^` is equivalent to lookaround `(?!.*)`. Accordingly, we can conclude that adding a lookaround/start-of-line anchor ensures that γ contains the same type of construct as δ which overlaps with another construct in sub-regex α or β, and the construct in γ overlaps with the same construct in α or β too.

Therefore, if δ contains a construct overlapping with another construct in sub-regex α or β, then γ contains this type of construct too, which overlaps with the same construct in α or β. □

Lemma 3. *Given a ReDoS-vulnerable regex $r = \alpha\gamma\beta$ where γ is the pathological sub-regex, RegexScalpel returns the repaired regex $r' = \alpha\delta\beta$ where δ is a safe regex obtained by repairing γ with our repair patterns and $\mathcal{L}(\delta) \subseteq \mathcal{L}(\gamma)$. r' satisfies that no vulnerabilities in the form of our vulnerable patterns will be newly introduced.*

Proof. Suppose the contrary, that is, a new vulnerability in the form of our vulnerable patterns will be introduced in r', which means that there are new nested quantifiers or overlapping constructs (i.e., overlapping disjunctions or adjacencies) in r'. Apparently, there are no nested quantifiers in the safe regex δ, so no nested quantifiers will be newly introduced in r'. Therefore, there are new overlapping constructs in r'. It's obvious that δ contains the construct which overlaps with another construct in sub-regex α or β. According to Lemma 2, we can conclude that γ contains the same construct as δ which overlaps with another construct in sub-regex α or β. Accordingly, this vulnerability must exist in r too, meaning that the vulnerability is not newly introduced, which leads to a contradiction. Hence we can conclude that no vulnerabilities in the form of our vulnerable patterns will be newly introduced in r'. □

Theorem 2. *Given a ReDoS-vulnerable regex r, RegexScalpel returns the repaired regex r' satisfying that there are no vulnerabilities in the form of our vulnerable patterns.*

Proof. By Lemma 3, it can be proved that no vulnerabilities will be newly introduced in r'. Then the iterative repair method can ensure that the final repaired regex does not contain any vulnerable patterns. □