VULCANBOOST: Boosting ReDoS Fixes through Symbolic Representation and Feature Normalization

Yeting Li[†§*], Yecheng Sun[†§*], Zhiwu Xu[‡], Haiming Chen[♮], Xinyi Wang[†§], Hengyu Yang[†§], Huina Chao[†§✉], Cen Zhang[¶], Yang Xiao[†§], Yanyan Zou[†§], Feng Li[†§], Wei Huo[†§✉]
[†] *Institute of Information Engineering, Chinese Academy of Sciences, China*
[§]*School of Cyber Security, University of Chinese Academy of Sciences, China*
[‡]*College of Computer Science and Software Engineering, Shenzhen University, China*
[♮]*Institute of Software, Chinese Academy of Sciences, China*
[¶]*School of Computer Science and Engineering, Nanyang Technological University, Singapore*

Abstract

Regular expressions (regexes) are widely used in modern programming languages but are susceptible to ReDoS attacks due to the inefficiencies introduced by backtracking algorithms. Existing approaches for repairing ReDoS-vulnerable regexes struggle with supporting diverse character classes and extended features, often relying on test cases for repair guidance. In this paper, we introduce VULCANBOOST, a novel framework for repairing ReDoS-vulnerable regexes that addresses these challenges. VULCANBOOST leverages symbolic representation and feature normalization to simplify regex structures and repair them through DFA (Deterministic Finite Automaton) transformations, eliminating the need for test case-based repair. Our evaluation, conducted on a large dataset of 6,360 ReDoS-vulnerable regexes from real-world NPM projects, demonstrates that VULCANBOOST achieves a Test Coverage Repair Success Rate (TCRSR) of 93.95% and an Equivalence Repair Success Rate (ERSR) of 93.05%, outperforming existing methods. Moreover, we identify common vulnerability patterns from over 5,000 repaired regexes and summarize the top 100 repair patterns as open-source resources, offering valuable guidance to developers in enhancing the security and correctness of their regexes.

1 Introduction

Regular expressions (regexes) are a fundamental component of nearly all modern programming languages, valued for their simplicity and efficiency in expressing complex pattern matching and text processing logic. They are extensively utilized across diverse fields in computer science, including network protocol analysis [48], intrusion detection systems [6,32], crawlers [18], databases [2], and text editors [29]. Studies [8,13,20] indicate that approximately 30% to 40% of JavaScript and Python projects incorporate at least one regex.

[*]Equal Contribution.
[✉] Corresponding Authors.

The widespread adoption of regexes has brought not only convenience but also serious security concerns—most notably, their vulnerability to a specific type of denial-of-service attack known as Regular Expression Denial of Service (ReDoS) [12, 16, 35]. To achieve expressiveness and flexibility, most regex engines—such as those embedded in Java, Python, Perl, Ruby, and JavaScript—employ backtracking algorithms (*e.g.*, Spencer's algorithm [37]), which can be easily extended with advanced features (also called *extended features*) like lookarounds and backreferences. However, like a double-edged sword, the application of backtracking algorithms also introduces potential security vulnerabilities—the super-linear worst-case matching time relative to the input length, making these engines vulnerable to catastrophic backtracking, namely ReDoS attacks. Recent statistics [13, 20] indicate that up to 10% of the regexes used in open-source software projects are vulnerable to ReDoS, thereby significantly increasing the risk of supply chain attacks [3]. In fact, recent years have witnessed a series of high-profile ReDoS attacks, with notable examples including those targeting Stack Overflow [38] and Cloudflare [7].

❑ **Existing Approaches.** Current defenses against ReDoS attacks largely fall into two categories. The first replaces inefficient regex engines with faster alternatives. While this improves performance, it often comes at the cost of functionality—advanced features such as lookarounds and backreferences are frequently unsupported, rendering many regexes unusable. Additionally, engine substitution may introduce semantic discrepancies, leading to mismatched results or incompatibilities with existing code, which can undermine system stability and reliability [5].

The second strategy focuses on repairing vulnerable regexes directly. Compared to engine substitution, regex repair is overwhelmingly preferred in mitigating ReDoS vulnerabilities due to its low cost, simplicity, and ease of maintaining compatibility. In fact, it is the most commonly used repair strategy, accounting for over 90% of recent ReDoS-related vulnerabilities [22]. Existing regex repair approaches predominantly leverage the conventional *example-guided* repair method. While state-of-the-art approaches (*e.g.*, Remedy [10], and RegexScalpel [22]) alleviate the burden on practitioners to repair vulnerable regexes, they mostly suffer from two major limitations below.

❶ **Insufficient Support for Diverse Character Classes in Real-World Regexes.** Real-world regexes often involve diverse character classes—also referred to as character sets—including predefined classes (*e.g.*, `\d` for digit characters and `\D` for non-digit characters), Unicode classes (*e.g.*, `\p{L}` for letters), and POSIX classes (*e.g.*, `[[:space:]]` for whitespace characters). However, existing repair approaches typically offer limited or incomplete support for these complex character classes. Compounding the challenge, such classes frequently correspond to extremely large alphabets—for example, the predefined complement class `\D`, which matches all non-digit characters, spans 65,526 characters. This dramatically expands the search space, making symbolic exploration or enumeration-based repair computationally infeasible within a reasonable time frame.

❷ **Lack of Examples in Real-World Regexes.** Existing approaches primarily rely on examples (also called test cases) as repair guidelines. However, in real-world projects, a common fact is that regexes are often not accompanied by corresponding test cases. An empirical study indicates that, in real-world project development, approximately 80% of regexes lack associated test cases; furthermore, among those that do have test cases, nearly half contain only a single test string [45]. Consequently, this undermines the practical effectiveness of existing example-driven approaches, particularly in complex scenarios where comprehensive test cases are absent.

❑ **Insight and Solution.** To address the aforementioned limitations, in this paper, we propose a regex repair algorithm, VULCANBOOST[1], which leverages two simple yet effective insights. *The first insight involves symbolizing regexes, which helps simplify the structure of the original regex and reduces the complexity of problem-solving.* Take, for example, the vulnerable regex `\w+\d+`, which is commonly used to match identifiers with numeric suffixes (*e.g.*, `user123`, `item42`). We decompose `\w` into its constituent parts—`\d` (digits) and `[A-Za-z_]` (letters and underscores)—and assign them symbolic labels, such as `a` for`\d` and `b` for `[A-Za-z_]`. This transforms the original regex into a symbolic form like `[ab]+a+`. The repair process then operates on this abstracted version—*e.g.*, converting `[ab]+a+` into the safer form `[ab]+a`. Afterward, the symbolic expressions are mapped back to their original character classes, yielding a repaired regex such as `\w+\d`. This approach not only avoids the direct manipulation of large and diverse character classes, but also significantly reduces the size of the alphabet (in this case, to just two symbols), thereby effectively addressing Limitation ❶.

The second insight involves transforming the vulnerable regex into a Deterministic Finite Automaton (DFA), followed by applying the state-elimination method to derive a repaired regex. This approach theoretically ensures that the repaired regex is equivalent to the original one. Furthermore, the entire repair process does not rely on specific examples, thus eliminating the dependence on test cases and, in turn, addressing Limitation ❷. Additionally, DFAs help us avoid ambiguities and backtracking during the matching process, thereby effectively mitigating ReDoS [42]. For instance, the vulnerable regex `(\d+(,\d+))+`, which is often used to match comma-separated lists of numbers (*e.g.*, `1,2,3`). Using our approach, it can be repaired into the safer variant `\d(,\d|\d)*`, which maintains the intended matching behavior while significantly improving efficiency and resilience against ReDoS attacks.

❑ **Challenges.** However, there exists several challenges to realize the insights. The first challenge arises from the need to reduce the problem size and enhance solution efficiency by using fewer symbols. *Specifically, the challenge lies in how to automatically and efficiently determine the minimum number of symbols required to represent the pairwise disjoint subsets of character classes in a regex, especially when multiple character classes and large alphabets are involved.* To address this, we first reduce the problem to solving the Venn diagram of character classes within the regex, where the number of non-empty regions in the Venn diagram corresponds to the minimum number of symbols required. Through this approach, we can confirm that the number of symbols obtained is minimal. Through analyzing numerous vulnerable regexes, we found that their character classes often exhibit subset and mutual exclusion relationships. Leveraging these relationships, we propose an effective heuristic strategy to efficiently prune empty regions in the Venn diagram, quickly calculating the minimum number of symbols required.

Due to the inadequacy of existing DFA-based tools in supporting extended features, the second challenge arises: *how can we implement our DFA-based repair algorithm to effectively handle regexes that include extended features?* To address this challenge, we systematically analyzed and semantically modeled ten mainstream extended features (such as anchors and lookarounds, etc.) and found that their semantics typically involve further restrictions on matching certain characters or positions, based on the original standard regexes. Based on this observation, we reformulate the problem of converting regexes with extended features into the intersection of DFAs corresponding to multiple standard regexes. For example, the regex `(?=a)\w+` describes strings that are matched by `\w+` and must start with the character 'a'. Therefore, the automaton corresponding to `(?=a)\w+` is equivalent to the intersection of the automaton for `\w+` and the automaton for `a`Σ^*. Please note that non-regular features, such as backreferences, are theoretically unsupported by DFA-based solutions. However, our findings indicate that pathological sub-regexes frequently do not pertain to these features.

❑ **Implementation and Evaluation.** To this end, we imple-

[1]VULCANBOOST derives its name from Vulcan, the Roman god of fire, combined with "Boost", symbolizing the tool's goal of enhancing the effectiveness of ReDoS repair.

mented VULCANBOOST as a regex repair framework and evaluated its performance against existing ReDoS repair methods using a large-scale dataset of 6,360 real-world ReDoS-vulnerable regexes from NPM projects.

- We present VULCANBOOST, a framework for repairing ReDoS-vulnerable regexes. To the best of our knowledge, VULCANBOOST is the first approach that leverages both symbolic representation and feature normalization to effectively fix vulnerable regexes with diverse character classes and mainstream extended features, while ensuring that the repair process does not rely on examples.
- The evaluation shows that VULCANBOOST significantly outperforms all baseline tools, with a Test Coverage Repair Success Rate (TCRSR) of 93.95% and an Equivalence Repair Success Rate (ERSR) of 93.05%, effectively mitigating ReDoS vulnerabilities without compromising correctness.
- By analyzing over 5,000 repaired regexes, we systematically identified recurring vulnerability patterns and summarized the top 100 repair patterns. These patterns will be released as an open-source resource (available at [23]), providing valuable guidance for users to avoid common mistakes and enhance the security and accuracy of regexes. They are particularly useful for users with less experience, offering important reference points.

2 Background and Preliminaries

Before introducing VULCANBOOST, we first review the theoretical foundations relevant to our approach, including regexes, ReDoS, and DFA. Let Σ denote a finite alphabet. The set of all words over Σ is represented by Σ^*. The empty word and empty set are denoted by ε and $\emptyset$, respectively. $\mathbb{N}$ denotes natural numbers including zero.

2.1 Regular Expressions (Regexes)

The formal syntax of a regex r over Σ is given as follows:

$$\begin{aligned} r \quad ::= \quad & [C] \mid \varepsilon \mid rr \mid r|r \mid r^{\{m,n\}} \mid (r)_i \mid \backslash i \\ & \mid (?<name>\ r) \mid \backslash k<name> \mid (?:\ r) \\ & \mid (?=r) \mid (?!r) \mid (?<=r) \mid (?<!r) \\ & \mid \hat{}r \mid r\$ \mid r_1\backslash b\, r_2 \mid r_1\backslash B\, r_2 \end{aligned}$$

A regex r is a structured expression that may include standard operators—concatenation, alternation, and Kleene star—as well as extended features such as capturing groups, non-capturing groups, named groups, backreferences, lookarounds, and anchors. Quantifiers $r^{\{m,n\}}$ match a pattern r between m and n times, where $m,n \in \mathbb{N}$ and $m \leq n$. For simplicity and clarity, 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 $r^{\{m,\}}$.

Extended regex features are clarified through detailed explanations and examples. A *capturing group* (r) stores the matched substring, indexed by i in $(r)_i$. For instance, `Set(Value)?` matches "`Set`" or "`SetValue`", where the capturing group is empty in the former and matches "`Value`" in the latter. These captured values can be accessed using a numerical backreference like $\backslash i$, which refers to the first group. In contrast, a *non-capturing group* $(?{:}r)$ avoids capturing, as in `Set(?:Value)?`. Long regexes with numerous groups can be hard to read and maintain, but named capturing groups, accessible by name, address this issue. In JavaScript, `(?<name>re)` captures re with a named reference name, where the name is alphanumeric and can be any regex.

Backreference $\backslash i$ matches the text captured by the group with index i. For example, `([abc])=\1` matches "`a=a`", "`b=b`", and "`c=c`", where the first group captures the character and `\1` matches the same character. This ensures that the same content appears in both positions. Similarly, named capturing group `(?<name>re)` can be referenced using `\k<name>`.

Lookarounds are zero-width assertions that enforce contextual constraints without consuming characters—they simply assert whether a match is allowed at a given position, and are divided into lookahead and lookbehind. Positive lookahead $(?=r_1)r_2$ matches r_2 only if preceded by r_1, while negative lookahead $(?!r_1)r_2$ ensures r_2 is not preceded by r_1. Lookbehind also has positive $r_1(?<=r_2)$, which matches r_1 only if preceded by r_2, and negative $r_1(?<!r_2)$, which ensures r_1 is not preceded by r_2. For example, the regex `(?=a)\w+` matches strings that consist of word characters `\w+` and must begin with the character "`a`".

Anchors are also zero-width assertions. The start-of-line anchor `^` marks the line's beginning, while`$` marks its end. The word boundary anchor `\b` matches positions where a word transitions to a non-word character, or at the string's start/end if they are word characters. Conversely, the non-word boundary anchor `\B` matches where `\b` does not. For example, `.\b` matches "`a@`" but not "`ab`".

2.2 Regex Denial of Service (ReDoS)

ReDoS is an attack exploiting algorithmic complexity in regex engines. A regex r is ReDoS-vulnerable on regex engine if there exists a string w such that the engine cannot decide whether $w \in \mathcal{L}(r)$ in $O(|w|)$ time[2]. For example, the regex `(a+)+b` can be exploited with the input `a...a`, causing exponential growth in matching time as the input size increases, disrupting service for legitimate users.

2.3 Deterministic Finite Automata (DFA)

Deterministic finite automaton (DFA) [17] is the finite automaton that ensures a single, unique transition exists for

[2] $\mathcal{L}(r)$ denotes the language accepted by the regex r.

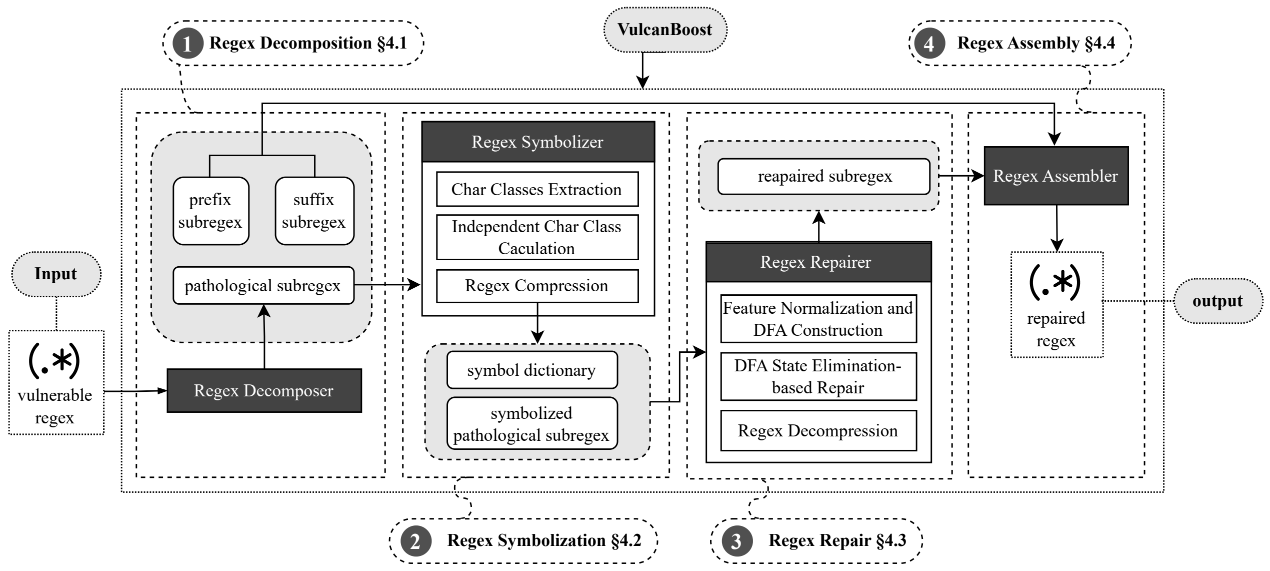

Figure 1: An Overview of VULCANBOOST for ReDoS Repair.

each input symbol at any given state. Formally, DFA is a quintuple $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set called the *set of states*, Σ is a finite set called *the input alphabet*, $q_0 \in Q$ is the *initial/start* state, $F \subseteq Q$ is the set of *final/accept states*, and $\delta : Q \times \Sigma \longrightarrow Q$ is a function called the *transition function* or *next-state function*.

A DFA operates as follows: Starting at the initial state q_0, the input device reads one symbol at a time from left to right. If the automaton ends in a final state from F, the string is accepted; otherwise, it is rejected. The transition function δ defines the movement between states, such that for each state and input symbol, δ assigns a unique next state. For example, $\delta(q_0, a) = q$ indicates that if the DFA is in state q_0 and reads the symbol 'a', it transitions to state q.

3 Overview

In this section, we present the core idea of our approach through a motivating example and explain how the key components of VULCANBOOST collaborate to achieve an end-to-end repair workflow for ReDoS-vulnerable regexes.

Example 1. Consider the vulnerable regex $\mathcal{R}$ = `^␣*(?!\t).+$`, which matches lines that begin with optional spaces, followed by a character that is not a tab. This pattern is typically used to enforce indentation rules that disallow mixing spaces and tabs. Maliciously crafted inputs like '␣'.repeat(n) + '`\n`' can induce catastrophic backtracking, leading to a ReDoS attack by exhausting CPU resources.

To systematically address such issues, we propose an automated repair framework called VULCANBOOST, whose overall workflow is illustrated in Figure 1. The framework consists of four key components: *regex decomposer*, *regex symbolizer*, *regex repairer*, and *regex assembler*. These components form a fully automated pipeline that repairs vulnerable regexes into semantically equivalent, ReDoS-resilient versions—without requiring any input examples.

The four components of our framework interact as follows. The *regex decomposer* first splits the input regex into prefix, pathological, and suffix subregexes. The *regex symbolizer* then extracts character classes from the pathological subregex, analyzes their relationships (*e.g.*, intersection and difference), partitions them into disjoint character subclasses, and assigns a symbolic token to each, yielding a symbolized regex along with a symbol dictionary. The *regex repairer* builds a DFA for the symbolized regex; if extended features are present, feature normalization is applied by modeling their semantics via the intersection of multiple DFAs. The repaired symbolic regex is then derived through state elimination and desymbolized using the symbol dictionary. Finally, the *regex assembler* combines the repaired subregex with the original prefix and suffix to produce the final repaired regex.

We demonstrate each step through a ReDoS-vulnerable regex presented in Example 1, with its systematic repair process depicted in Figure 2. **Step ❶ Regex Decomposition.** The *regex decomposer* begins by statically identifying the pathological portion of the original regex $\mathcal{R}_{inf}$ = `␣*(?!\t).+`. It then decomposes the original regex into: prefix subregex $\mathcal{R}_{pre}$ = `^`, pathological subregex $\mathcal{R}_{inf}$ and suffix subregex $\mathcal{R}_{suf}$ = `$`. This decomposition helps isolate the vulnerable part for focused repair while preserving the semantic correctness of the surrounding context (see Step ❶ in Figure 2).

Step ❷ Regex Symbolization. The *regex symbolizer* takes the pathological subregex $\mathcal{R}_{inf}$ as input and identifies the char-

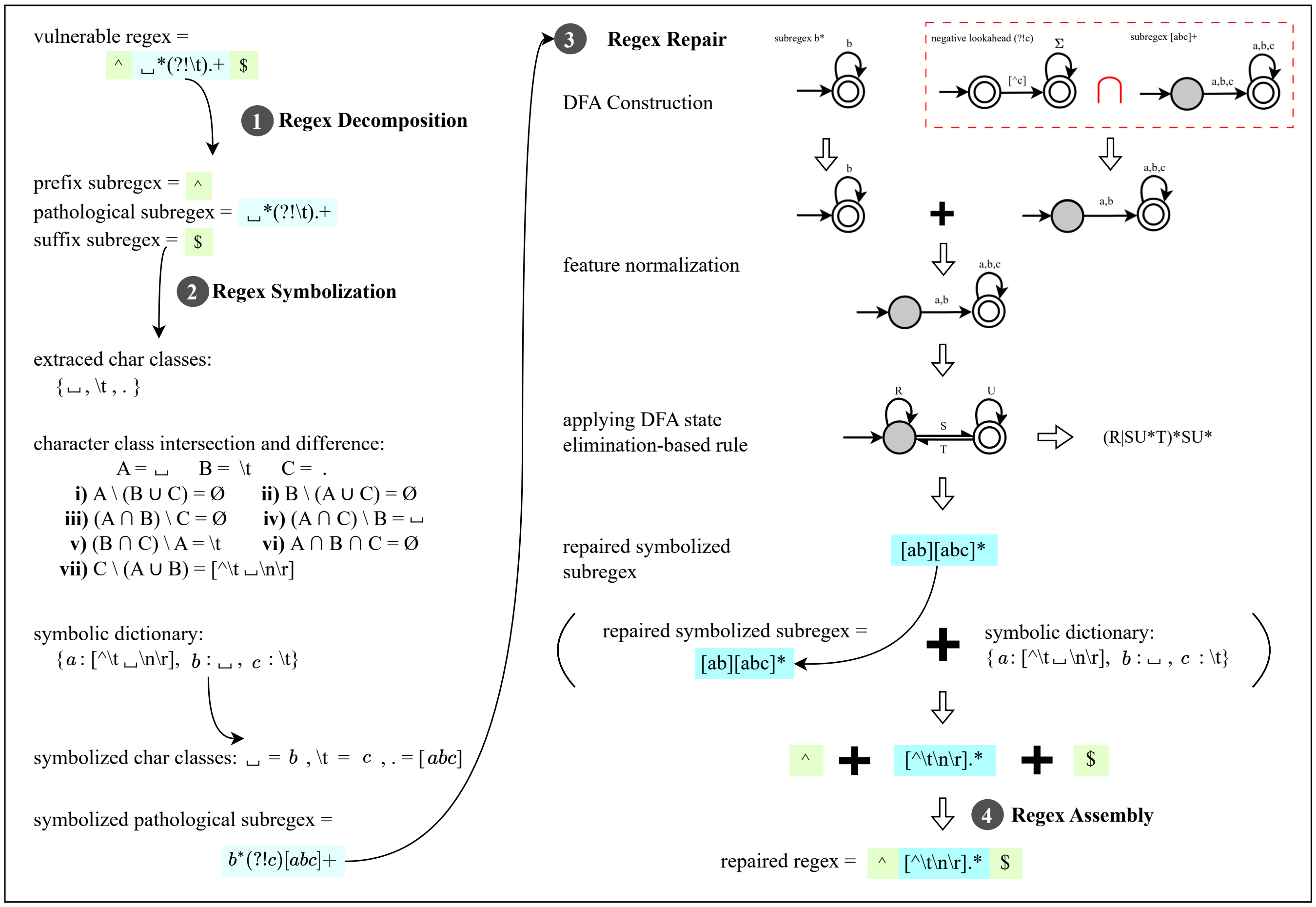

Figure 2: An Example of VULCANBOOST.

acter classes involved, including $A =$ ␣ (space), $B =$ `\t` (tab), and $C =$ `.` (dot). Set analysis is performed over these classes to compute non-empty intersections and differences, which are then partitioned into three disjoint character classes: $C \setminus (A \cup B) =$ `[^\t␣\n\r]`, $(A \cap C) \setminus B =$ ␣, and $(B \cap C) \setminus A =$ `\t`. Each disjoint class is assigned a symbolic token—specifically, `[^\t␣\n\r]` is mapped to `a`, ␣ to `b`, and `\t` to `c`. The pathological subregex $\mathcal{R}_{inf} =$ `␣*(?!\t).+` is thus transformed into a symbolized form $\mathcal{R}^{\P}_{inf} =$ `b*(?!c)[abc]+` . A symbol dictionary ¶ is constructed in parallel to record the mapping between each symbolic token and its corresponding character class, enabling desymbolization in the subsequent repair step (see Step ❷ in Figure 2).

Step ❸ Regex Repair. The *regex repairer* takes the symbolized regex $\mathcal{R}^{\P}_{inf}$ as input and decomposes its semantic components into three DFAs: $\mathcal{M}_1$ representing the subregex `b*`, $\mathcal{M}_2$ corresponding to the negative lookahead `(?!c)`, whose semantics exclude strings beginning with the symbol `c`, and $\mathcal{M}_3$ representing the subregex `[abc]+`. These three automata first undergo feature normalization; specifically, VULCANBOOST computes the intersection of DFA $\mathcal{M}_2$ and DFA $\mathcal{M}_3$, and then concatenates this intersection after DFA $\mathcal{M}_1$, thereby constructing a unified DFA $\mathcal{M}$. Then, VULCANBOOST applies the state-elimination method on $\mathcal{M}$ to derive a symbolic regex $\mathcal{R}^{\P}_{inf}{}'$ = `[ab][abc]*`, which is semantically equivalent to $\mathcal{R}^{\P}_{inf}$. It is important to note that DFAs help us avoid ambiguities and backtracking during the matching process, thereby effectively mitigating the triggering of ReDoS vulnerabilities. Finally, through desymbolization, VULCANBOOST recovers the final repaired subregex $\mathcal{R}_{inf}{}' =$ `[^\t\n\r].*`, with a detailed procedure illustrated in Step ❸ of Figure 2.

Step ❹ Regex Assembly. In the final step, the *regex assembler* concatenates the repaired pathological subregex $\mathcal{R}_{inf}{}'$ with the original prefix and suffix subregexes (*i.e.*, $\mathcal{R}_{pre}$ and $\mathcal{R}_{suf}$) to construct the final repaired regex $\mathcal{R}' =$ `^[^\t\n\r].*$`. The resulting regex $\mathcal{R}'$ is free from vulnerabilities; for example, when matching the aforementioned malicious string '␣'.repeat(n) + '\n', it operates linearly without triggering catastrophic backtracking. Furthermore, this regex is semantically equivalent to the original regex $\mathcal{R}$. Notably, the entire repair process took only 1.91 seconds and is independent of specific examples.

4 Methodology

4.1 Regex Decomposition

Regex decomposition isolates the pathological subcomponents of a vulnerable regex to enable precise, localized repair and reduce the computational overhead of symbolization and transformation. Unlike prior approaches such as Remedy [10], which operate on the entire regex and often result in *over-repair*—unintended modifications to benign segments—our method preserves structural and semantic integrity, improving both readability and maintainability.

Guided by the ReDoS vulnerability patterns proposed by Wang et al. [46], we analyze regexes under the *exact match* setting, where the pattern must match the entire input string[3]. The vulnerable regex is then split into prefix, pathological, and suffix segments for targeted processing.

4.2 Regex Symbolization

Regex symbolization simplifies pathological subregexes by abstracting complex structures and large character classes into symbolic representations. The process extracts character classes from the regex, partitions their overlaps into mutually disjoint subsets—conceptually akin to a Venn diagram—and assigns each subset a unique symbolic token. These tokens replace the original character classes, yielding a symbolized subregex and a symbol dictionary for later desymbolization.

To clarify the algorithmic design, we begin with a straightforward solution that highlights the core idea and exposes its limitations. Section 4.2.1 presents the naive approach, while Section 4.2.2 introduces an optimized version that addresses these inefficiencies.

4.2.1 Naive Symbolization

As a baseline, we introduce a naive algorithm (Algorithm 1) that enumerates all character class combinations in a regex to derive mutually disjoint symbolic tokens, thereby supporting structured downstream processing. The algorithm proceeds as follows:

1. **Character Class Extraction:** Parse the input regex $\mathcal{R}$ into an abstract syntax tree (AST) and extract all character classes, yielding $\mathbb{C} = \{c_1, c_2, \ldots, c_N\}$ (line 1), where each c_i denotes a concrete character class (*e.g.*, `[a-z]`).

2. **Venn Diagram Partitioning:** Construct a complete Venn diagram partition $\mathbb{R}$ over the set $\mathbb{C}$ (line 2) by enumerating all bipartitions $(\mathbb{C}_{in}, \mathbb{C}_{out})$ such that

$$\mathbb{C}_{in} \cup \mathbb{C}_{out} = \mathbb{C}, \quad \mathbb{C}_{in} \cap \mathbb{C}_{out} = \emptyset,$$

   representing unique regions in the Venn diagram.

3. **Non-empty Subset Computation:** For each bipartition, compute the corresponding character subset

$$\mathcal{S} = \left(\bigcap_{\alpha \in \mathbb{C}_{in}} \alpha \right) \setminus \left(\bigcup_{\beta \in \mathbb{C}_{out}} \beta \right)$$

   If $\mathcal{S} \neq \emptyset$, add it to the partition set Δ (lines 4–5).

4. **Symbol Assignment and Regex Rewriting:** Assign a unique symbolic token to each subset in Δ to form a symbol dictionary $\P$. Replace all original character classes in $\mathcal{R}$ with their corresponding tokens to obtain the symbolic regex $\mathcal{R}^{\P}$ (line 6).

[3]The exact match setting refers to requiring the regex to match the entire input string, as in `re.fullmatch()` in Python or regexes wrapped with `^...$` in JavaScript.

Algorithm 1: Naive Regex Symbolization

```
Input: A regex $\mathcal{R}$
Output: A symbolized regex $\mathcal{R}^{\P}$, a symbol dictionary $\P$
1 $\mathbb{C} \leftarrow$ extract_character_classes($\mathcal{R}$)
2 $\mathbb{R} \leftarrow$ create_venn_regions($\mathbb{C}$)
3 $\Delta \leftarrow [\,]$
4 foreach $(\mathbb{C}_{in}, \mathbb{C}_{out}) \in \mathbb{R}$ such that $\mathbb{C}_{in} \cap \mathbb{C}_{out} = \emptyset$ and
  $\mathbb{C}_{in} \cup \mathbb{C}_{out} = \mathbb{C}$ do
5     $\Delta \overset{+}{\leftarrow} (\bigcap_{\alpha \in \mathbb{C}_{in}} \alpha) \setminus (\bigcup_{\beta \in \mathbb{C}_{out}} \beta)$, if non-empty
6 return $\mathcal{R}^{\P}, \P \leftarrow$ symbolize_regex($\mathcal{R}, \Delta$)
```

Example 2. To illustrate the above procedure, consider the input regex:

$$\mathcal{R} = \texttt{[0-469]+[1-35-7][1-58]+}$$

The algorithm begins by parsing $\mathcal{R}$ into an AST and extracting the set of character classes:

$$\mathbb{C} = \{c_1 = \texttt{[0-469]},\ c_2 = \texttt{[1-35-7]},\ c_3 = \texttt{[1-58]}\}$$

with their corresponding character sets:

$$\begin{aligned} c_1 &: \{0,1,2,3,4,6,9\} \\ c_2 &: \{1,2,3,5,7\} \\ c_3 &: \{1,2,3,4,5,8\} \end{aligned}$$

Next, the algorithm constructs a complete Venn diagram partition $\mathbb{R}$ over the set $\mathbb{C}$ and enumerates all bipartitions $(\mathbb{C}_{in}, \mathbb{C}_{out})$, as illustrated in Figure 3. It then computes candidate subclasses by taking intersections of the character classes in $\mathbb{C}_{in}$ and subtracting the union of those in $\mathbb{C}_{out}$. This process yields the following mutually disjoint, non-empty character subsets:

$$\Delta = \left\{ \begin{array}{ll} c_1 \setminus (c_2 \cup c_3) & : \{0,9\}, \\ c_2 \setminus (c_1 \cup c_3) & : \{7\}, \\ c_3 \setminus (c_1 \cup c_2) & : \{8\}, \\ (c_1 \cap c_2) \setminus c_3 & : \{6\}, \\ (c_1 \cap c_3) \setminus c_2 & : \{4\}, \\ (c_2 \cap c_3) \setminus c_1 & : \{5\}, \\ (c_1 \cap c_2 \cap c_3) \setminus \emptyset & : \{1,2,3\} \end{array} \right\}$$

The algorithm then define a symbolic alphabet ¶ by assigning a unique symbol to each region in Δ:

$$¶ = \left\{ \begin{array}{l} \texttt{a} \to \{0,9\}, \\ \texttt{b} \to \{7\}, \\ \texttt{c} \to \{8\}, \\ \texttt{d} \to \{6\}, \\ \texttt{e} \to \{4\}, \\ \texttt{f} \to \{5\}, \\ \texttt{g} \to \{1,2,3\} \end{array} \right\}$$

By substituting the character classes in $\mathcal{R}$ with their corresponding symbolic letters, the algorithm derives the symbolic regex:

$$\mathcal{R}^{¶} = \texttt{[adeg]+[bdfg][cefg]+}$$

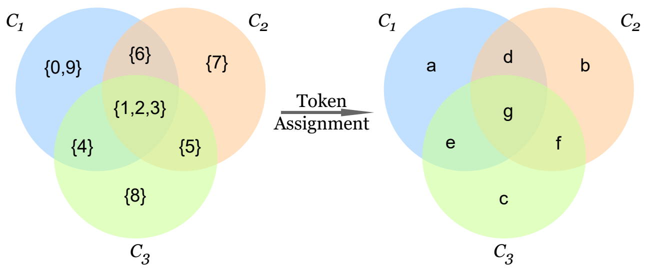

Figure 3: Character Classes Venn Diagram illustrating the seven disjoint subsets derived from the character classes $\mathbb{C} = \{c_1 = \texttt{[0-469]}, c_2 = \texttt{[1-35-7]}, c_3 = \texttt{[1-58]}\}$, each mapped to a unique symbolic token.

4.2.2 Optimized Symbolization via Structural Pruning

Given a set of character classes $\mathbb{C} = \{C_1, C_2, \ldots, C_N\}$, the naive approach requires enumerating all $2^N - 1$ subsets to construct candidate non-empty regions in the character space (*e.g.*, as in a Venn diagram). This is computationally expensive, especially when N is large.

In practice, structural relationships among character classes—in particular, *disjointness* and *subset inclusion*—can be leveraged to significantly reduce the search space. The proposed optimization (Algorithm 2) introduces a static preprocessing step to compute two key sets:

Structural Relationships. We define two precomputed binary relations over $\mathbb{C}$:

- $\mathbb{P}_{dis} \subseteq \mathbb{C} \times \mathbb{C}$: the *disjoint-pair set*, where $(c_i, c_j) \in \mathbb{P}_{dis}$ if $c_i \cap c_j = \emptyset$;
- $\mathbb{P}_{sub} \subseteq \mathbb{C} \times \mathbb{C}$: the *subset-pair set*, where $(c_i, c_j) \in \mathbb{P}_{sub}$ if $c_i \subseteq c_j$.

Algorithm 2: Optimized Regex Symbolization with Conflict Detection

```
Input: A regex $\mathcal{R}$
Output: A symbolized regex $\mathcal{R}^{¶}$, a symbol dictionary ¶
1  Function is_conflict_pair($\mathbb{C}_{in}, \mathbb{C}_{out}, \mathbb{P}_{dis}, \mathbb{P}_{sub}$):
2      if $\exists\, \alpha, \beta \in \mathbb{C}_{in}$ s.t. $(\alpha, \beta) \in \mathbb{P}_{dis}$ then
3          return true
4      else if $\exists\, \alpha \in \mathbb{C}_{in}, \beta \in \mathbb{C}_{out}$ s.t. $(\alpha, \beta) \in \mathbb{P}_{sub}$ then
5          return true
6      return false
7  $\mathbb{C} \leftarrow$ extract_character_classes($\mathcal{R}$)
8  $\mathbb{P}_{dis} \leftarrow$ compute_disjoint_pairs($\mathbb{C}$)
9  $\mathbb{P}_{sub} \leftarrow$ compute_subset_pairs($\mathbb{C}$)
10 $\mathbb{R} \leftarrow$ create_venn_regions($\mathbb{C}$)
11 $\Delta \leftarrow [\,]$
12 foreach $(\mathbb{C}_{in}, \mathbb{C}_{out}) \in \mathbb{R}$ such that $\mathbb{C}_{in} \cap \mathbb{C}_{out} = \emptyset$ and
       $\mathbb{C}_{in} \cup \mathbb{C}_{out} = \mathbb{C}$ do
13     if is_conflict_pair($\mathbb{C}_{in}, \mathbb{C}_{out}, \mathbb{P}_{dis}, \mathbb{P}_{sub}$) then
           continue ;
14     else $\Delta \xleftarrow{+} \left(\bigcap_{\alpha \in \mathbb{C}_{in}} \alpha\right) \setminus \left(\bigcup_{\beta \in \mathbb{C}_{out}} \beta\right)$ , if non-empty ;
15 return $\mathcal{R}^{¶}$, ¶ $\leftarrow$ symbolize_regex($\mathcal{R}, \Delta$)
```

Conflict Detection. During bipartition enumeration of $(\mathbb{C}_{in}, \mathbb{C}_{out})$, we define a function `is_conflict_pair` which returns `true` if any of the following conditions hold:

1. $\exists\,(c_i, c_j) \in \mathbb{P}_{dis}$, s.t. $c_i, c_j \in \mathbb{C}_{in}$ (internal disjointness);
2. $\exists\,(c_i, c_j) \in \mathbb{P}_{sub}$, s.t. $c_i \in \mathbb{C}_{in}$, $c_j \in \mathbb{C}_{out}$ (cross-subset violation).

If either condition holds, the bipartition cannot yield a valid non-empty character region and is thus skipped.

Pruned Region Enumeration. Rather than computing all $2^N - 1$ candidate regions, the algorithm enumerates only conflict-free bipartitions, each yielding a valid symbolic subset:

$$\left(\bigcap_{\alpha \in \mathbb{C}_{in}} \alpha\right) \setminus \left(\bigcup_{\beta \in \mathbb{C}_{out}} \beta\right),$$

if and only if $\neg$`is_conflict_pair`$(\mathbb{C}_{in}, \mathbb{C}_{out}, \mathbb{P}_{dis}, \mathbb{P}_{sub})$.

Example 3. Consider the regex $\mathcal{R} = $ `\s\d\w`, where the character classes correspond to: `\s` (whitespace), `\d` (digits), and `\w` (word characters).

The structural relations are:

$$\mathbb{P}_{dis} = \{(\texttt{\textbackslash s}, \texttt{\textbackslash d}),\ (\texttt{\textbackslash s}, \texttt{\textbackslash w})\}, \quad \mathbb{P}_{sub} = \{(\texttt{\textbackslash d}, \texttt{\textbackslash w})\}.$$

Using these relations:

- The three candidates ($\mathbb{C}_{in} = \{\texttt{\textbackslash s}, \texttt{\textbackslash d}\}$, $\mathbb{C}_{out} = \{\texttt{\textbackslash w}\}$), ($\mathbb{C}_{in} = \{\texttt{\textbackslash s}, \texttt{\textbackslash w}\}$, $\mathbb{C}_{out} = \{\texttt{\textbackslash d}\}$), and ($\mathbb{C}_{in} = \{\texttt{\textbackslash s}, \texttt{\textbackslash d}, \texttt{\textbackslash w}\}$, $\mathbb{C}_{out} = \emptyset$) are rejected due to disjoint conflict.

- The candidate ($\mathbb{C}_{in} = \{\texttt{\d}\}$, $\mathbb{C}_{out} = \{\texttt{\s}, \texttt{\w}\}$) is rejected due to subset conflict.

Among the 7 regions generated by the naive method, only 3 remain (*i.e.*, ($\mathbb{C}_{in} = \{\texttt{\s}\}$, $\mathbb{C}_{out} = \{\texttt{\d}, \texttt{\w}\}$), ($\mathbb{C}_{in} = \{\texttt{\w}\}$, $\mathbb{C}_{out} = \{\texttt{\s}, \texttt{\d}\}$), and ($\mathbb{C}_{in} = \{\texttt{\d}, \texttt{\w}\}$, $\mathbb{C}_{out} = \{\texttt{\s}\}$)) valid under the pruning strategy. In general, identifying P disjoint pairs can reduce the region search space from exponential 2^N to $(P+1)\cdot 2^{N-P}$, substantially reducing computational overhead.

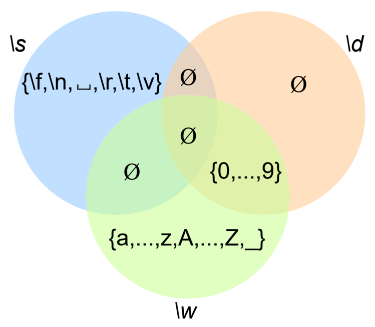

Figure 4: Three valid Venn regions after pruning conflicting combinations among character classes \s, \d, and \w based on disjoint and subset constraints.

4.3 Regex Repair

This section introduces the *repair* algorithm for mitigating ReDoS vulnerabilities. It leverages the DFA model to transform regexes into automata and back. The algorithm operates without examples and yields repaired regexes with enhanced resilience to ReDoS attacks. The detailed procedure is presented in Algorithm 3.

Since existing DFA-based tools lack native support for several widely-used extended features–such as lookaheads and non-capturing groups—we first conduct a systematic analysis and semantic modeling of ten commonly used regex extensions. These include positive lookahead, negative lookahead, positive lookbehind, negative lookbehind, start-of-line anchor, end-of-line anchor, word boundary anchor, non-word boundary anchor, non-capturing group, and named capturing group.

4.3.1 Lookaround Normalization

To accurately model the semantics of lookaround assertions (*i.e.*, positive and negative lookahead and lookbehind) in regex, we normalize each lookaround construct into automaton-theoretic operations over DFAs. Specifically, we define a transformation function regex2dfa(·) that maps a regex into its equivalent DFA, and utilize standard automata operations such as intersection (∩) and complement ($\bar{\cdot}$) to formalize lookaround semantics.

Algorithm 3: Regex Repair

```
Input: a symbol dictionary ¶, a symbolized regex $\mathcal{R}^{¶}$
Output: a regex $\mathcal{R}'$
Function regex2dfa($\mathcal{R}$):
    if $\mathcal{R} == (? = \mathcal{R}_1)\mathcal{R}_2$ then
        $\mathcal{A}_1 \leftarrow$ regex2dfa($\mathcal{R}_2$)
        $\mathcal{A}_2 \leftarrow$ regex2dfa($\mathcal{R}_1\Sigma^*$)
        return $\mathcal{A}_1 \cap \mathcal{A}_2$
    else if $\mathcal{R} == (?!\mathcal{R}_1)\mathcal{R}_2$ then
        $\mathcal{A}_1 \leftarrow$ regex2dfa($\mathcal{R}_2$)
        $\mathcal{A}_2 \leftarrow \overline{\text{regex2dfa}(\mathcal{R}_1\Sigma^*)}$
        return $\mathcal{A}_1 \cap \mathcal{A}_2$
    else if $\mathcal{R} == \mathcal{R}_1(? <= \mathcal{R}_2)$ then
        $\mathcal{A}_1 \leftarrow$ regex2dfa($\mathcal{R}_1$)
        $\mathcal{A}_2 \leftarrow$ regex2dfa($\Sigma^*\mathcal{R}_2$)
        return $\mathcal{A}_1 \cap \mathcal{A}_2$
    else if $\mathcal{R} == \mathcal{R}_1(? <!\mathcal{R}_2)$ then
        $\mathcal{A}_1 \leftarrow$ regex2dfa($\mathcal{R}_1$)
        $\mathcal{A}_2 \leftarrow \overline{\text{regex2dfa}(\Sigma^*\mathcal{R}_2)}$
        return $\mathcal{A}_1 \cap \mathcal{A}_2$
    else if $\mathcal{R} == \hat{}\,\mathcal{R}_1$ then
        return regex2dfa($(? <!\Sigma)\mathcal{R}_1$)
    else if $\mathcal{R} == \mathcal{R}_1\$$ then
        return regex2dfa($\mathcal{R}_1(?!\Sigma)$)
    else if $\mathcal{R} == \mathcal{R}_1\backslash b\mathcal{R}_2$ then
        $\mathcal{A}_1 \leftarrow$ regex2dfa($\mathcal{R}_1(?<=\backslash w)(?=\backslash W)\mathcal{R}_2$)
        $\mathcal{A}_2 \leftarrow$ regex2dfa($\mathcal{R}_1(?<=\backslash W)(?=\backslash w)\mathcal{R}_2$)
        return $\mathcal{A}_1 \cup \mathcal{A}_2$
    else if $\mathcal{R} == \mathcal{R}_1\backslash B\mathcal{R}_2$ then
        $\mathcal{A}_1 \leftarrow$ regex2dfa($\mathcal{R}_1(?<=\backslash w)(?=\backslash w)\mathcal{R}_2$)
        $\mathcal{A}_2 \leftarrow$ regex2dfa($\mathcal{R}_1(?<=\backslash W)(?=\backslash W)\mathcal{R}_2$)
        return $\mathcal{A}_1 \cup \mathcal{A}_2$
    else if $\mathcal{R} == (? : \mathcal{R}_1)$ then
        return regex2dfa($\mathcal{R}_1$)
    else if $\mathcal{R} == (?P<name>\mathcal{R}_1)$ then
        return regex2dfa($\mathcal{R}_1$)
    else
        return std_regex2dfa($\mathcal{R}$);      // standard
         regex to DFA conversion
$\mathcal{M}^{¶} \leftarrow$ regex2dfa($\mathcal{R}^{¶}$)
$\mathcal{R}^{¶'} \leftarrow$ dfa2regex($\mathcal{M}^{¶}$)
$\mathcal{R}^{¶'} \leftarrow$ simplify($\mathcal{R}^{¶'}$)
return $\mathcal{R}' \leftarrow$ desymbolize($\mathcal{R}^{¶'}$, ¶)
```

Let Σ denote the input alphabet, and Σ^* its Kleene closure. The formal semantics are defined as follows:

- **Positive Lookahead**: For $\mathcal{R} = (?=\mathcal{R}_1)\mathcal{R}_2$, the match succeeds only if the input at the current position has a prefix that matches $\mathcal{R}_1$, before continuing with $\mathcal{R}_2$. This is captured by:

$$\text{regex2dfa}(\mathcal{R}) = \text{regex2dfa}(\mathcal{R}_2) \cap \text{regex2dfa}(\mathcal{R}_1\Sigma^*)$$

- **Negative Lookahead**: For $\mathcal{R} = (?!\mathcal{R}_1)\mathcal{R}_2$, the assertion requires that the input at the current position does *not* have a prefix matching $\mathcal{R}_1$. This is captured by:

$$\text{regex2dfa}(\mathcal{R}) = \text{regex2dfa}(\mathcal{R}_2) \cap \overline{\text{regex2dfa}(\mathcal{R}_1\Sigma^*)}$$

- **Positive Lookbehind**: For $\mathcal{R} = \mathcal{R}_1(?<=\mathcal{R}_2)$, the match succeeds only if the input *prior* to the current position has a suffix that matches $\mathcal{R}_2$. This is formalized by:

$$\text{regex2dfa}(\mathcal{R}) = \text{regex2dfa}(\mathcal{R}_1) \cap \text{regex2dfa}(\Sigma^*\mathcal{R}_2)$$

- **Negative Lookbehind**: For $\mathcal{R} = \mathcal{R}_1(?<!\mathcal{R}_2)$, the assertion requires that the input before the current position does *not* end with a substring matching $\mathcal{R}_2$. This is expressed as:

$$\text{regex2dfa}(\mathcal{R}) = \text{regex2dfa}(\mathcal{R}_1) \cap \overline{\text{regex2dfa}(\Sigma^*\mathcal{R}_2)}$$

This reduction uniformly maps all lookaround semantics to DFA intersection and complementation, as implemented in Algorithm 3, lines 1–17.

4.3.2 Anchor Normalization

Anchors such as the start-of-line ^, end-of-line $, word boundary \b, and non-word boundary \B can be equivalently expressed using lookahead and lookbehind assertions.

- **Start-of-Line Anchor (^)**: For the regex $\hat{}\mathcal{R}_1$, we transform it into a lookbehind assertion that ensures no prefix precedes the match:

$$\text{regex2dfa}(\hat{}\mathcal{R}_1) = \text{regex2dfa}((?<!\Sigma)\mathcal{R}_1)$$

- **End-of-Line Anchor ($)**: For the regex $\mathcal{R}_1\$$, we transform it into a lookahead assertion that ensures no suffix follows:

$$\text{regex2dfa}(\mathcal{R}_1\$) = \text{regex2dfa}(\mathcal{R}_1(?!\Sigma))$$

- **Word Boundary (\b)**: For the regex $\mathcal{R}_1\backslash b\mathcal{R}_2$, we use both lookahead and lookbehind to ensure a word boundary between $\mathcal{R}_1$ and $\mathcal{R}_2$:

$$\begin{aligned}\text{regex2dfa}(\mathcal{R}_1\backslash b\mathcal{R}_2) = {} & \text{regex2dfa}(\mathcal{R}_1(?<=\backslash w)(?=\backslash W)\mathcal{R}_2) \\ & \cup \text{regex2dfa}(\mathcal{R}_1(?<=\backslash W)(?=\backslash w)\mathcal{R}_2)\end{aligned}$$

- **Non-Word Boundary (\B)**: Similarly, for the regex $\mathcal{R}_1\backslash B\mathcal{R}_2$, we assert the absence of a word boundary:

$$\begin{aligned}\text{regex2dfa}(\mathcal{R}_1\backslash B\mathcal{R}_2) = {} & \text{regex2dfa}(\mathcal{R}_1(?<=\backslash w)(?=\backslash w)\mathcal{R}_2) \\ & \cup \text{regex2dfa}(\mathcal{R}_1(?<=\backslash W)(?=\backslash W)\mathcal{R}_2)\end{aligned}$$

This transformation, as implemented in lines 18-29 of Algorithm 3, standardizes anchor handling and ensures that the regex2dfa function can process these constructs uniformly and effectively.

4.3.3 Group Normalization

To handle group constructs in regex, we normalize both non-capturing and named capturing groups into subexpression DFAs. These groups differ from the previous extended features in that they do not impose semantic constraints like lookarounds or anchors, but rather serve syntactic purposes such as improving readability or controlling capturing behavior. We unify their handling as follows:

- **Non-Capturing Group**: For $(?:\mathcal{R}_1)$, the transformation is identical, except without capturing annotations:

$$\text{regex2dfa}((?:\mathcal{R}_1)) = \text{regex2dfa}(\mathcal{R}_1)$$

- **Named Capturing Group**: For $(?P<name>\mathcal{R}_1)$, we preserve capturing semantics by directly mapping to the sub-expression:

$$\text{regex2dfa}((?P<name>\mathcal{R}_1)) = \text{regex2dfa}(\mathcal{R}_1)$$

This normalization streamlines the handling of groups at the automaton level. See Algorithm 3, lines 30–33.

4.3.4 DFA-based Regex Repair and Simplification

After systematically handling extended regex features, the repair algorithm proceeds by invoking the function regex2dfa to convert the input symbolized regex $\mathcal{R}^\P$ into its equivalent DFA $\mathcal{M}^\P$ (line 36). Subsequently, leveraging the classical DFA state elimination method [17], $\mathcal{M}^\P$ is converted back into a repaired regex $\mathcal{R}^{\P'}$ (line 37). This process effectively eliminates ambiguities and backtracking inherent in the original regex, thus mitigating the risk of ReDoS attacks.

The advantage of this approach is twofold: First, the use of DFA enforces determinism and clarity in the matching process. Second, due to prior regex decomposition, the DFA representations of pathological subregexes remain compact, minimizing the chance of introducing new ambiguities during the reverse transformation.

To improve the readability and conciseness of the repaired regex $\mathcal{R}^{\P'}$, we apply a set of simplification rules (line 38), including: (1) removing redundant parentheses that do not affect semantics (*e.g.*, `a(b)c` becomes `abc`); (2) merging adjacent characters and quantifiers into more compact forms (*e.g.*, `aa*` becomes `a+`); and (3) eliminating empty disjuncts (*i.e.*, ε) in alternations (*e.g.*, `(a|ε)` becomes `a?`).

Finally, desymbolization is performed by replacing abstract symbols in $\mathcal{R}^{\P'}$ with their original character classes according to the symbol dictionary ¶ established during symbolization (line 39). The resulting repaired regex $\mathcal{R}'$ is semantically equivalent to the original but exhibits improved readability, conciseness, and resilience against ReDoS vulnerabilities.

4.4 Regex Assembly

The *regex assembly* combines the prefix subregex $\mathcal{R}_{pre}$ (obtained from *regex decomposition*), the suffix subregex $\mathcal{R}_{suf}$ (also obtained from *regex decomposition*), and the repaired subregex $\mathcal{R}'$ (obtainied from *regex repair*) to complete the repair process.

5 Implementation and Evaluation

We developed the VULCANBOOST prototype to repair regexes vulnerable to ReDoS attacks. The tool VULCANBOOST is implemented using over 1,200 lines of non-comment Python code and 3,500 lines of non-comment Java code. It integrates multiple existing tools while independently developing critical modules (*e.g.*, the *Regex Symbolizer*). The primary modules of VULCANBOOST are as follows:

❏ **Regex Decomposition.** VULCANBOOST builds upon the *Rengar* [46] vulnerability detection tool and the *ANTLR4* [31] parser generator to enable the segmentation of original regexes. This process identifies pathological subregexes (*i.e.*, vulnerable components prone to ReDoS attacks) and isolates them for further analysis and repair, ensuring a targeted and efficient approach to addressing regex vulnerabilities.

❏ **Character Class Extraction.** Leveraging Python's *sre_parse* [1] module, VULCANBOOST extracts character classes from the isolated pathological subregexes, providing essential semantic information for subsequent analysis.

❏ **Regex Symbolization.** The Regex Symbolizer, a core component independently implemented in VULCANBOOST, is designed to simplify pathological subregexes by addressing challenges inherent in regex repair, such as the complexity of regexes and large character classes. By computing the intersections and differences of character classes, the symbolization process reduces the structural complexity of the subregex while preserving its essential semantics, thereby enabling more efficient repair.

❏ **Regex Repair.** VULCANBOOST utilizes a custom-developed extended feature processor to semantically model the extended features in pathological subregexes. It then employs the *dk.brics* [30] automaton library to construct the intersection automaton corresponding to these extended features. Then, the *autorex* [41] utility, which implements state elimination techniques, is used to convert the intersection automaton into an equivalent regex. Finally, using our custom-built regex simplifier, we simplify the deterministic regex obtained from *autorex*, yielding an optimized regex.

In this section, we present an experimental evaluation of VULCANBOOST to address the following research questions:

- **RQ1**: How does the repair effectiveness of VULCANBOOST compare to that of SOTA methods for mitigating ReDoS vulnerabilities? (§5.2)
- **RQ2**: How does the repair efficiency of VULCANBOOST compare to that of SOTA methods for mitigating ReDoS vulnerabilities? (§5.3)
- **RQ3**: Does VULCANBOOST benefit from the application of augmentation techniques, and how do they contribute to its overall performance? (§5.4)

5.1 Experiment Setup

❏ **Benchmark Selection.** To ensure a fair and unbiased evaluation of the repair capabilities of various tools, we adopted a consistent approach for selecting ReDoS-vulnerable regex patterns. Given that the Remedy tool exclusively addresses ReDoS vulnerability repair under exact matches, and to ensure a more objective comparison, this paper similarly limits its analysis to the repair of ReDoS vulnerabilities in the context of exact matches. This approach also serves as the selection criterion for our benchmark.

To conduct the evaluation, we collected a large set of ReDoS-vulnerable regex patterns from real-world projects. First, we utilized the *all-the-package-repos*[4] package to obtain the repository URLs of JavaScript modules and downloaded several JavaScript module projects based on these URLs. Subsequently, we employed the regex extraction tool developed by Davis et al. [13], which identifies and extracts regex patterns from these projects using an Abstract Syntax Tree (AST). In total, we extracted 80,000 regex patterns.

For these extracted regexes, we employed the ReDoS detection tool *Rengar* [46] to identify potential ReDoS-vulnerable regexes. Based on the attack strings generated by the tool, we implemented a dynamic validation script to filter out regex patterns potentially vulnerable to ReDoS attacks. As a result, we identified 6,360 ReDoS-vulnerable regexes.

Since existing repair tools rely on test examples during execution, we randomly generated training and testing datasets for each of the 6,360 regexes. Specifically, we created 5 examples for the training set, following the requirements of tool Remedy [10], and an additional 5 examples for the testing set

[4] https://www.npmjs.com/package/all-the-package-repos

to evaluate performance. The details of these regexes together with their examples can be found online [23].

❑ **Comparative Tools.** We evaluated VulcanBoost in comparison with Remedy [10] and RegexScalpel [22], two SOTA techniques for repairing ReDoS-vulnerable regexes. i) Remedy, which is an example-driven tool for repairing ReDoS-vulnerable regexes, supporting extended features like lookarounds and backreferences. ii) RegexScalpel, which is a tool that fixes ReDoS-vulnerable regexes based on predefined repair templates, using example-based validation to select the appropriate template.

❑ **Evaluation Metrics.** To assess the effectiveness of VulcanBoost, the following evaluation metrics are employed:

- **Test Coverage Repair Success Rate (TCRSR):** The ratio of the number of repairs that pass all test cases and are free from ReDoS attacks to the total number of regexes under repair.
- **Equivalence Repair Success Rate (ERSR):** The ratio of the number of repairs that are equivalent to the original regexes and are free from ReDoS attacks to the total number of regexes under repair.

To implement these metrics, the following procedures were followed:

- **ReDoS Vulnerability Check:** For verifying whether a repaired regex contains any ReDoS vulnerabilities, we used the Rengar tool. The verification process closely mirrors the benchmark construction process.
- **Test Case Coverage:** The test coverage for each repaired regex was evaluated using a pre-constructed benchmark testing set. We used a dynamic validation script to run the repaired regexes against the test cases to verify whether they passed successfully.
- **Equivalence Verification:** To assess whether the repaired regex is equivalent to the original one, we followed a multi-step process. ❶ Initial Verification with Test Cases. If the repaired regex fails any test case, it is classified as non-equivalent to the original. ❷ Automated Equivalence Check. If the regex passes the test cases, we then used the *dk.brics* [30] library to check if the original regex and the repaired regex are semantically equivalent. ❸ Symbolization and Manual Review. In cases where *dk.brics* cannot handle the regex pair, we resorted to symbolization and used them to perform equivalence checks. If symbolization still doesn't yield a conclusion, the final step was manual validation. In this process, the three authors independently reviewed the cases, and any inconsistencies were discussed and resolved until a consensus was reached.

Table 1: **The Effectiveness of Regex Repair.** The symbols #Sol, #CSol, #Equ and #Vul represent the following for each tool: the total number of repairs, the number of repairs that pass all test cases, the number of repairs that are equivalent to the original regexes, and the number of vulnerabilities remaining after repair.

Technique	#Sol	#CSol	#Equ	#Vul	TCRSR	ERSR
Remedy [10]	1,683	1,220	114	26	1,210(19.03%)	114(1.79%)
RegexScalpel [22]	4,180	2,621	321	1,369	1,389(21.84%)	54(0.85%)
VulcanBoost	**5,975**	**5,975**	**5,918**	**0**	**5,975(93.95%)**	**5,918(93.05%)**

❑ **Configurations.** The experiments were conducted on a machine equipped with a 13th Gen Intel(R) Core(TM) i9-13900KF CPU, operating at 3.00GHz with 24 cores and 36MB of cache, complemented by 64GB of RAM and running Windows 11. For all experiments, we established the following time thresholds: the regex decomposition time threshold, the regex symbolization time threshold, and the regex repair time threshold. Each threshold was set to 60 seconds. All baseline configurations were set according to the settings reported in the original papers.

5.2 Effectiveness Evaluation (RQ1)

❑ **Overall Results.** The comparative analysis (Table 1) demonstrates the outstanding performance of VulcanBoost in repairing ReDoS-vulnerable regexes. We evaluate the repair success using two metrics: the Test Coverage Repair Success Rate (TCRSR), which is a more lenient indicator, and the Equivalence Repair Success Rate (ERSR), which is a stricter measure. In both aspects, VulcanBoost significantly outperforms the other tools, achieving TCRSR and ERSR values of 93.95% and 93.05%, respectively. In comparison, the second-best tool, RegexScapel, shows a TCRSR of 21.84% and an ERSR of 0.85%, while Remedy, the least effective tool, achieves a TCRSR of 19.03% and an ERSR of 1.79%.

These results highlight the substantial advantage of VulcanBoost in terms of repair success rate and equivalence of the repairs. To further elaborate on these findings, we analyze the fine-grained metrics of #Sol, #CSol, #Equ, and #Vul, which offer deeper insight into the repair effectiveness and quality of each tool.

❑ **Total Number of Repairs (#Sol).** The #Sol metric reflects the total number of repairs conducted by each tool. Among the 6,360 vulnerable regexes, VULCANBOOST produces the most repairs—5,975—which is 1.43× the number produced by RegexScalpel (4,180) and 3.55× that of Remedy (1,683). This demonstrates VULCANBOOST's superior repair coverage and capability.

❑ **Repairs Passing All Test Cases (#CSol) and Equivalent (#Equ).** The #CSol metric measures the number of repairs that pass all test cases, while #Equ represents the number of repairs that are equivalent to the original regexes. In both of these metrics, VULCANBOOST excels. Among its 5,975 repairs, 100% pass all test cases, and 99.05% (5,918) are semantically equivalent to the original regexes. While 57 repairs are not equivalent due to engineering limitations in its *autorex*-based [41] implementation, this represents only a small fraction.

In comparison, RegexScalpel's repairs are less comprehensive: only 2,621 out of 4,180 (62.70%) pass all test cases, and only 321 (7.68%) are equivalent. Remedy, while demonstrating some effectiveness, shows more limited results, with 1,220 (72.49%) test-passing repairs and 114 (6.77%) equivalence. In contrast, VULCANBOOST significantly outperforms both baselines in terms of test coverage and semantic equivalence. This notable improvement highlights VULCANBOOST's robustness and practical utility, particularly in scenarios with limited test examples.

❑ **Repair Failure Analysis for VULCANBOOST.** Although VULCANBOOST significantly improves the overall repair success rate over existing approaches, 385 regexes remain unrepaired. A detailed analysis reveals two primary contributing factors. First, the repair process for certain complex regexes exceeds the predefined time limit, resulting in timeouts. This issue accounts for 200 cases, comprising 51.95% of the unrepaired instances. Second, while VULCANBOOST supports a wide range of character classes and extended features, there are still certain character classes that it cannot repair. Specifically, 185 (48.05%) regexes, fall into this category. Examples include regexes containing Unicode characters such as `\x{10FFFF}`.

❑ **Repair Failure Analysis for Remedy.** The poor repair performance of the Remedy tool can be attributed to three main reasons. First, it does not support certain simple character classes, such as `\s`, `\S`, `\d`, `\D`, `\w`, and `\W`. For example, the regex `^-?\d*\.?\d*%` cannot be repaired. Second, Remedy lacks support for extended features like `\b` and `\B`. For instance, the regex `\bObject\.?<([^,]*), *([^>]*)>` cannot be handled. Finally, even after repair, some vulnerabilities remain. For example, the regex `([a-zA-Z0-9._-]+\.[a-zA-Z0-9._-]+)` is repaired into `((?:(?:[^-.28CJTZ_gijtuy])+).(?:(?:[a-zA-Z0-9._-])+))` by Remedy, but it still exhibits vulnerabilities, which can still be exploited in a ReDoS attack by the input string '`\n`' + '00'×30,000 + '`\n`'. In addition to having vulnerabilities, the repair overly relies on known examples, resulting in a significant deviation from the original semantics.

❑ **Repair Failure Analysis for RegexScalpel.** The failure of repair in RegexScalpel can largely be attributed to the absence of corresponding repair templates for many pathological subexpressions. Specifically, for certain vulnerable regexes, such as `^[a-z0-9]+([.-][a-z0-9]+)*@([a-z0-9]+([.-][a-z0-9]+))+$`, there are no available templates for repair, resulting in an inability to fix the issues. On the other hand, even when a repair is applied, vulnerabilities may still persist. For example, the vulnerable regex `.*DELETE_THIS_LINE.*(\r|\n)`, after repair, becomes `^.{0,500}DELETE_THIS_LINE.*(\r|\n)`, yet the vulnerability remains. Attackers can exploit this by using a specially crafted input string (such as, '`DELETE_THIS_LINET`'$\times *$30000+ '`␣EEE...`') to trigger the flaw, posing a security risk.

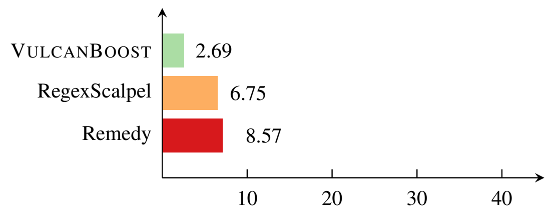

Figure 5: Comparison on Speed with STOA Tools (Seconds).

Summary to RQ1: VULCANBOOST outperforms all baseline tools with a Test Coverage Repair Success Rate (TCRSR) of 93.95% and an Equivalence Repair Success Rate (ERSR) of 93.05%. It successfully repairs 5,975 regexes, covering a substantial portion of ReDoS vulnerabilities, and demonstrates its ability to deliver high-quality repairs with 99.05% equivalence.

5.3 Efficiency Evaluation (RQ2)

❑ **Comparison of Repair Times.** In the comparison of repair times among regex repair tools, VULCANBOOST demonstrates clear superiority, achieving the shortest average repair time of 2.69 seconds. In contrast, RegexScalpel and Remedy exhibit longer average repair times of 6.75 seconds and 8.57 seconds, respectively. It is important to highlight that both RegexScalpel and Remedy fail to repair a substantial number of regexes, which inadvertently reduces their average repair time. Specifically, Remedy provides repairs for 1,683 regexes, RegexScalpel for 4,180, while VULCANBOOST successfully repairs 5,975 regexes. Consequently, the average repair time of VULCANBOOST is not only efficient but also more representative and stable across a broader set of inputs.

Summary to RQ2: VULCANBOOST achieves the shortest average repair time of 2.69 seconds, outperforming RegexScalpel (6.75 seconds) and Remedy (8.57 seconds). In addition to its high repair coverage (5,975 regexes), it offers a compelling balance between efficiency and effectiveness, underscoring its competitive advantage in practical scenarios.

Table 2: **Comparison for VULCANBOOST Variants.**

Technique	#Sol	#Equ	#Vul	ERSR	Time (s)
VULCANBOOST $_{wrdo}$	5,097	5,038	0	5,038(79.21%)	10.77
VULCANBOOST $_{autorex}$	5,925	4,523	0	4,523(71.12%)	0.74
VULCANBOOST	**5,975**	**5,918**	**0**	**5,918(93.05%)**	2.69

5.4 Ablation Studies (RQ3)

To evaluate the effectiveness of the proposed techniques in addressing the limitations outlined earlier, we conducted a series of ablation studies by selectively disabling key components of VULCANBOOST and analyzing their individual contributions. Specifically, we implemented two variants: i) VULCANBOOST $_{wrdo}$, which disables the symbolization optimization and employs only the naive regex symbolization algorithm; ii) VULCANBOOST $_{autorex}$, which adopts a basic DFA-based state-elimination algorithm for regex conversion, without support for extended regex features.

❏ **Contribution of Symbolization Optimization.** To assess the role of symbolization optimization, we compared VULCANBOOST against the variant VULCANBOOST $_{wrdo}$. VULCANBOOST accelerates symbolization by precisely capturing disjointness and subset relationships among character class intersections and differences. As a result, VULCANBOOST successfully repaired 878 additional regexes compared to VULCANBOOST $_{wrdo}$, leading to a 13.84% improvement in the ERSR. Furthermore, the average repair time decreased from 10.77 seconds to 2.69 seconds, achieving a 4.00× speedup.

❏ **Contribution of Feature Support.**

To assess the benefit of extended feature support, we compared VULCANBOOST with the variant VULCANBOOST $_{autorex}$, which lacks support for advanced regex constructs. By accurately modeling ten major extended features (*e.g.*, anchors and lookarounds), VULCANBOOST was able to repair 1,395 additional regexes that VULCANBOOST $_{autorex}$ failed to handle. This enhancement increased the ERSR from 71.12% to 93.05%. Although the average repair time increased by 1.95 seconds, this is attributable to the additional processing required to handle extended features. Given the substantial gain in repair success, the additional time overhead is deemed acceptable.

Summary to RQ3: The evaluation demonstrates the effectiveness of each component of VULCANBOOST. Symbolization optimization led to 878 additional repairs, enhancing ERSR by 13.84% and reducing repair time by 4.00×. Extended feature support enabled 1,395 more repairs, boosting ERSR by 21.93%, with a increase in repair time by 1.95 seconds.

6 Lessons Learned

"It's good to learn from your mistakes. It's better to learn from other people's mistakes."

—Warren Buffett

Through the analysis of over five thousand repaired regexes in our experiments, we observed several noteworthy phenomena. First, although regexes written by different authors vary in structure and intended usage, the types of errors they contain often follow similar patterns. Second, certain vulnerability patterns appear with disproportionately high frequency, indicating that these common pitfalls merit focused attention to improve regex robustness.

For instance, the regexes `[-0-9A-Z_a-z]+[-0-9:A-Z_a-z]*` and `[-0-9:A-Z_a-z]+[^>]*`, though semantically different and used in distinct contexts, both exhibit the same vulnerability pattern `a+[ab]*`. This can be mitigated by a more precise form, such as `a[ab]*`. Recognizing and reusing such repair patterns can help developers avoid common sources of vulnerability.

Motivated by these observations, we validated the repairs produced in our experiments and distilled the most recurring fixes into a collection of the top 100 repair patterns. These patterns cover a significant portion of the total repairs and reflect key strategies for mitigating frequent vulnerabilities. Due to space limitations, we present the top 20 in Table 3, while the complete set is available online [23].

We believe sharing these patterns with the open-source community can offer practical benefits, particularly for developers less experienced with regex. These patterns may serve not only as educational examples but also as a reference library for fixing existing regexes or addressing newly discovered issues.

7 Limitations and Future Improvements

The tool VULCANBOOST has certain limitations, which we aim to address in future improvements.

❏ **Limited Support for Character Classes.** First, while VULCANBOOST supports common character classes (*e.g.*, `\s`, `\d`, and `\p{}`), it does not support more complex classes like `\x{}`. Future work will extend support for additional character classes, improving extensibility.

❏ **Handling Non-Regular Extended Features.** Secondly, Although VULCANBOOST does not support non-regular extended features such as backreferences, its impact is minimal since backreferences rarely appear in pathological regexes, they are more common in sliced prefix and suffix subregexes (*e.g.*, `<(style|script).*?>.*?<\/\1>`). However, to enhance VULCANBOOST's extensibility, we propose two improvements: Firstly, for cases where backreferences involve

Table 3: **Top 20 Most Common Vuln. Patterns in Regexes.**

ID	Vuln. Pattern	Repair Pattern	Freq
1	`a+[ab]*`	`a[ab]*`	359
2	`[ab]*b[ab]*`	`a*b[ab]*`	263
3	`[ab]*[ac]*`	`[ab]*(c[ac]*)?`	234
4	`[ab]+[ac]*`	`[ab]+(c[ac]*)?`	209
5	`[ab]+[ac]+`	`[ab](a+b|b)*((a+c|c)[ac]*|a+)`	155
6	`[ab]+b[ab]+`	`[ab]a*b[ab]+`	150
7	`a*b?a*`	`a*(ba*)?`	134
8	`a+[ab]+`	`a[ab]+`	127
9	`a+b?a*`	`a+(ba*)?`	113
10	`a*b?a+`	`(a+b|b)a+|a+`	111
11	`a*[ab]*`	`[ab]*`	106
12	`a+b?a+`	`(aa+b|ab)a+|aa+`	87
13	`a*b*a*`	`a*(b+(a+)?)?`	86
14	`a+[ab]a+`	`a(a((a+b|b)a+|a+)|ba+)`	82
15	`[ab]*[ac]+`	`b*a(b+a|a)*(b+(c[ac]*)|c[ac]*)?|b*c[ac]*`	76
16	`[ab]*?b[ab]*?`	`a*b[ab]*`	75
17	`a*[ab]+`	`[ab]+`	73
18	`[ab]*a+`	`(a+b|b)*a+`	63
19	`[ab]*?b[ab]*`	`a*b[ab]*`	58
20	`a+[ab]*?`	`a[ab]*`	53

enumerable content, we suggest a method that unrolls simple backreferences to facilitate DFA analysis. For example, the regex `(['"])(.)\1` can be expanded into `'.'|".*"`, thereby improving the efficiency of DFA analysis. Secondly, future work will focus on developing models that support these extended features, particularly backreferences.

❑ **Equivalence of Repaired Regexes.** Finally, although the repaired regexes are theoretically equivalent to the original ones, limitations of third-party tool *autorex* [41] can cause discrepancies in certain cases. To mitigate this, we plan to introduce an equivalence verifier (such as using the *dk.brics* [30] library) to ensure the equivalence of the repaired regexes.

8 Related Work

❑ **Regex Repair.** Li et al. [24] proposed the pioneering programming-by-example (PBE) framework that leverages determinism for ReDoS repair via regex synthesis and repair. Despite its innovation, the framework lacks support for extended features (*e.g.*, lookarounds and backreferences). The ReDoS repair framework proposed by Chida et al. [9] employs a PBE approach too, featuring strong deterministic rules and compatibility with extended features such as lookarounds and backreferences. But both PBE works [9, 24] face a common issue: repair effectiveness relies heavily on user-provided examples, which are often difficult to supply, limiting the creation of effective regex repairs. Li et al. [22] introduced a tool that adopts a localized detection and repair methodology to pinpoint vulnerabilities using detailed patterns and implement predefined fixing techniques. However, the proposed repair patterns may alter the regex semantics and cannot cover all ReDoS vulnerabilities.

❑ **Regex Engine Optimization.** To combat ReDoS, researchers have proposed various regex engine optimizations, including Thompson's NFA [11, 43], counting automata [44], memoization [14], parallel processing [25], GPU-based approaches [50], state-merging [4], PEGs [15, 19, 28], and the like. These techniques improve engine efficiency and mitigate ReDoS attacks but may sacrifice memory or feature support.

❑ **Regex Detection.** Previous studies about ReDoS detection can be categorized into static analysis [33, 34, 47, 49], dynamic analysis [27, 36, 39, 40], and hybrid approaches combining static and dynamic analysis [21, 26]. Notably, three detectors (*i.e.*, [21, 26, 46]) have been developed to identify ReDoS vulnerabilities by formally modeling vulnerable patterns. While the patterns are comprehensive, they cannot support extracting pathological subregexes. Drawing inspiration from Wang et al. [46], we have presented the regex decomposition algorithm for subsequent repair.

9 Conclusion

In this paper, we present VULCANBOOST, a novel framework for repairing ReDoS-vulnerable regexes, addressing challenges in handling diverse character classes and extended features. By leveraging symbolic representation and feature normalization, VULCANBOOST simplifies the repair process and ensures compatibility with advanced regex features. Evaluation on 6,360 vulnerable regexes from real-world NPM projects shows VULCANBOOST achieves a Test Coverage Repair Success Rate (TCRSR) of 93.95% and an Equivalence Repair Success Rate (ERSR) of 93.05%. It repairs over 5,000 regexes with a high equivalence rate of 99.05%. In addition, we identify and release 100 recurring repair patterns distilled from successful cases, providing practical guidance for improving regex security and correctness in broader development contexts.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Key R&D Program of China (Grant No. 2022YFB3103900), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDCO2030200), the National Natural Science Foundation of China (Grant Nos. 62032010, 62202462, 62302500, 62372439, and 62372304), and the Beijing Natural Science Foundation (Grant Nos. 4232038 and F251021).

Ethics Considerations

In this research, we aim to enhance the security of regular expressions (regexes) in defending against ReDoS attacks through the development of the VULCANBOOST framework. While security analysis tools may be misused, we have implemented several precautionary measures to ensure the responsibility of this research. These measures include adhering to responsible vulnerability disclosure practices, coordinating with affected system maintainers, and providing detailed usage guidelines upon the open-source release of VULCANBOOST. We believe that the benefits of enhancing the security of regexes to protect millions of users far outweigh the potential risks, which have been effectively mitigated through the precautions we have taken.

Open Science

In accordance with USENIX Security's open science policy, we have made our tool, VULCANBOOST, publicly available to ensure the reproducibility of our research findings. The artifact includes the complete source code, documentation, and experimental setup. Additionally, we share the evaluation dataset used in our study, which contains the test cases and the top 100 high-value repair patterns identified from our experiments. All materials are archived and accessible via Zenodo at https://doi.org/10.5281/zenodo.15550469. We believe this open approach not only strengthens the transparency and credibility of our work, but also facilitates future research in repairing ReDoS vulnerabilities.

References

[1] Secret Labs AB. sre_parse, 2024. https://github.com/xbmc/python/blob/master/Lib/sre_parse.py.

[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Semant.*, 7(2):57–73, 2009.

[3] Efe Barlas, Xin Du, and James C. Davis. Exploiting Input Sanitization for Regex Denial of Service. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 883–895. ACM, 2022.

[4] 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.

[5] Masudul Hasan Masud Bhuiyan, Berk Çakar, Ethan H Burmane, James C Davis, and Cristian-Alexandru Staicu. Sok: A literature and engineering review of regular expression denial of service. *arXiv preprint arXiv:2406.11618*, 2024.

[6] João Bispo, Ioannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis. Regular Expression Matching for Reconfigurable Packet Inspection. In *2006 IEEE International Conference on Field Programmable Technology, FPT 2006, Bangkok, Thailand, December 13-15, 2006*, pages 119–126. IEEE, 2006.

[7] 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/.

[8] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 282–293, 2016.

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

[10] Nariyoshi Chida and Tachio Terauchi. Repairing DoS Vulnerability of Real-World Regexes. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2060–2077. IEEE, 2022.

[11] 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.

[12] Scott Crosby. Denial of service through regular expressions. 2003.

[13] 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.

[14] James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (redos). In *2021 IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, May 23-27, 2021*, page To appear, 2021.

[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] Jan Goyvaerts. Runaway regular expressions: Catastrophic backtracking, 2003.

[17] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd Edition*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[18] Jonathan M. Hsieh, Steven D. Gribble, and Henry M. Levy. The Architecture and Implementation of an Extensible Web Crawler. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 329–344. USENIX Association, 2010.

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

[20] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 415–426, 2019.

[21] 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.

[22] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 4183–4200. USENIX Association, 2022.

[23] Yeting Li, Yecheng Sun, Zhiwu Xu, Haiming Chen, Xinyi Wang, Hengyu Yang, Huina Chao, Cen Zhang, Yang Xiao, Yanyan Zou, Feng Li, and Wei Huo. Vulcanboost: Boosting redos fixes through symbolic representation and feature normalization, June 2025. https://doi.org/10.5281/zenodo.15550469.

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

[25] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. Accelerating regular expression matching using hierarchical parallel machines on GPU. In *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*, pages 1–5. IEEE, 2011.

[26] Y. Liu, M. Zhang, and W. Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1468–1484, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[27] 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.

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

[29] Robert C. Miller and Brad A. Myers. LAPIS: smart editing with text structure. In *Extended abstracts of the 2002 Conference on Human Factors in Computing Systems, CHI 2002, Minneapolis, Minnesota, USA, April 20-25, 2002*, pages 496–497. ACM, 2002.

[30] Anders Møller. dk.brics.automaton, 2024. https://www.brics.dk/automaton/.

[31] Terence Parrm. ANTLR, 2024. https://www.antlr.org.

[32] Jignesh Patel, Alex X. Liu, and Eric Torng. Bypassing Space Explosion in Regular Expression Matching for Network Intrusion Detection and Prevention Systems. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

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

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

[35] Alex Roichman and Adar Weidman. Vac-redos: Regular expression denial of service. *Open Web Application Security Project (OWASP)*, 2009.

[36] 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.

[37] Henry Spencer. *A regular-expression matcher*, pages 35–71. 1994.

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

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

[40] 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.

[41] Julian Thome. autorex: A dk.brics FSM to regular-expression-string converter, 2024. https://github.com/julianthome/autorex.

[42] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

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

[44] 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.

[45] Peipei Wang and Kathryn T Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 668–678, 2018.

[46] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, and Wei Huo. Effective redos detection by principled vulnerability modeling and exploit generation. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2427–2443. IEEE, 2023.

[47] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, pages 322–334, 2016.

[48] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. Automatic Network Protocol Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

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

[50] Xiaodong Yu and Michela Becchi. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In Hubertus Franke, Alexander Heinecke, Krishna V. Palem, and Eli Upfal, editors, *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 18:1–18:10. ACM, 2013.