Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks

Francesco Parolini, Antoine Miné

▶ **To cite this version:**

Francesco Parolini, Antoine Miné. Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks. 2022. hal-03685057

HAL Id: hal-03685057
https://hal.science/hal-03685057v1

Preprint submitted on 1 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks

Francesco Parolini and Antoine Miné

Sorbonne Université, CNRS, LIP6, 75005 Paris, France
{francesco.parolini, antoine.mine}@lip6.fr

Abstract. Modern programming languages often provide functions to manipulate regular expressions in standard libraries. If they offer support for advanced features, the matching algorithm has an exponential worst-case time complexity: for some so-called *vulnerable regular expressions*, an attacker can craft ad hoc strings to force the matcher to exhibit an exponential behaviour and perform a *Regular Expression Denial of Service* (ReDoS) attack. In this paper, we introduce a framework based on a tree semantics to statically identify ReDoS vulnerabilities. In particular, we put forward an algorithm to extract an *overapproximation* of the set of words that are dangerous for a regular expression, effectively catching all possible attacks. We have implemented the analysis in a tool called `rat`, and testing it on a dataset of 74,670 regular expressions, we observed that in 99.47% of the instances the analysis terminates in less than one second. We compared `rat` to four other ReDoS detectors, and we found that our tool is faster, often by orders of magnitude, than most other tools. While raising a low number of false positives, `rat` is the only ReDoS detector that does not report false negatives.

Keywords: Regular Expressions · Denial of Service · Algorithmic Complexity Attacks · Static Analysis · Security and Privacy.

1 Introduction

Regular expressions (regexes) are often used to verify that strings in programs match a given pattern. Modern programming languages offer support to regexes in standard libraries, and this encourages programmers to take advantage of them. However, matching engines of languages such as Python, JavaScript, and Java employ algorithms with exponential worst-case time complexity in the length of the string. This is because advanced features such as *backreferences* extend the expressiveness of regular expressions. This comes at the cost of exponential matching in the worst case, even for regexes that do not exploit such features. An attacker can craft a string to force the matcher to exhibit the exponential behaviour to perform a *Regular Expression Denial of Service* (ReDoS) attack, a particular type of *algorithmic complexity attack* [12].

ReDoS attacks are vastly underestimated Denial of Service (DoS) attacks. In a recent study of regexes usage, in nearly 4,000 Python projects on Github, the

authors find that over 42% of them contain regexes [9], while in [30] the authors found that 10% of the Node.js-based web services they examined are vulnerable to ReDoS. In this already harsh scenario, in [17] the authors find that only 38% of the developers that they surveyed knew about the *existence* of ReDoS attacks. Many well-known platforms observed such vulnerabilities in their systems: among them, we find Stack Overflow [29], Cloudflare [10], and iCloud [4]. Since it is difficult to detect ReDoS vulnerabilities with manual inspection, it is necessary to automate this critical process. However, for now, there is no practical and widely adopted solution to detect ReDoS vulnerabilities.

There are many different approaches to static semantics-based ReDoS detection [18,27,32,33], and they are all based on automata frameworks. Due to the difficulties to precisely model matching engines with automata, static analyzers often report both false positives and false negatives. In contrast, dynamic approaches to ReDoS detection [28] can hardly be used in practice, since performing dynamic testing on exponential algorithms can be excessively costly.

In this paper, we put forward a novel approach to statically detect ReDoS vulnerabilities. We get rid of the complexities to represent the behaviour of matching engines with automata by defining a *tree semantics* of the matching process. Next, we leverage it to introduce an analysis that determines whether a regex may be vulnerable or not. In particular, the analysis returns an *over-approximation* of the language of words that can cause exponential matching, being effectively *sound* but *not complete*. Nevertheless, our experiments show that our approach reports a low number of false positives.

In this work, we focus on the most dangerous type of ReDoS vulnerability, namely when the matching is exponential. To successfully perform an attack that exploits superlinear but non-exponential matching, a malicious user must be allowed to insert very large strings. Such attacks are considerably less dangerous than the case that we consider.

Our approach not only eliminates the complexities related to using automata, but also opens the possibility to easily introduce optimizations. We implemented our algorithm in a tool called `rat` [24], and we found it to be on average one to two orders of magnitude faster than most existing detectors, while being proved to be sound and raising only 50 false alarms over 74,670 regexes. Furthermore, `rat` can extract the language of possibly dangerous words, being strictly more expressive than most other tools. This expressiveness can be useful in different scenarios: for example, existing matching engines can use our algorithm to filter-out dangerous input strings. It is also possible to use the language of dangerous words by combining our framework with a string analysis in order to prove the absence of ReDoS vulnerabilities in real-world applications.

2 Background

2.1 ReDoS Vulnerabilities

The majority of programming languages that offer support for regexes in standard libraries are vulnerable to ReDoS attacks. Among them, we find Python,

```
1  import re
2  email_regex = r'^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@(([0-9
       a-zA-Z])+([-\w]*[0-9a-zA-Z])*\.)+[a-zA-Z]{2,9})$'
3  attack = 'a' * 50
4  re.match(email_regex, attack)
```

Fig. 1: Python program that matches a dangerous string against a vulnerable regex.

Java, JavaScript, PHP, and Ruby. Figure 1 shows an example of a Python program that matches a string with a vulnerable regex that validates email addresses. The regex is taken from the Regexlib [2] database, and possibly many programmers used it. Executing the program on a modern computer with a 4GHz Intel Core i7-4790K CPU takes more than 24 hours. In Section 4, we give in-depth description of ReDoS vulnerabilities, but here we informally introduce why this behaviour arises. Consider the input string a^{50} and the subexpression `([-.\w]*[0-9a-zA-Z])*`: a can be matched in `[-.\w]*` or in `[0-9a-zA-Z]`. This implies that in `([-.\w]*[0-9a-zA-Z])*` there are four paths to match aa , eight for aaa and in general 2^n for a^n. Normally, the matching engine accepts the first match, but here, as `@` does not appear in the string, it exhaustively explores all 2^{50} paths before concluding that no match is possible for a^{50} in the full regex.

Usually programming languages employ matching engines with exponential worst-time complexity to support advanced features such as *backreferences* and *lookarounds* [15,21]. We, like most other analyzers, do not support such features. Nevertheless, our approach is sufficient to analyze the great majority of regexes in real-world applications: in [9] the authors found that in nearly 4000 Python projects, only 4% of the regexes use lookarounds and up to 0.4% use backreferences. Yet, recent surveys determined that up to the 10% of the web services they considered present ReDoS vulnerabilities [30]. This highlights how programmers use vulnerable matching engines while only occasionally taking advantage of advanced features, and motivates the need for a sound ReDoS analyzer even limited to regular constructs.

2.2 ReDoS Detection

There are two main approaches to ReDoS detection:

1. *Semantics-based static detection.* There are many different approaches to semantics-based static ReDoS detection [18,26,27,32,33], and they all rely on automata. In those frameworks, regexes are first transformed into automata, which are then analyzed to determine whether they are vulnerable or not. The main problem is that transforming regexes to automata can remove or inject vulnerabilities. This is often a source of both false positives and false negatives. We discuss semantics-based static analyzers based on automata

in detail in Section 6, and we compare them to our approach that is also semantics-based, but operates on regexes instead of automata.

2. *Dynamic detection.* A dynamic analyzer generates strings that are fed to the matching engine. Then, the tool measures the time for the matching and determines whether a regex is vulnerable or not. These tools are sensibly slower than static analyzers, because performing testing on exponential algorithms can be excessively time-consuming. While it is possible to configure generic fuzzers, such as `SlowFuzz` [25], to detect ReDoS vulnerabilities, in [28] the authors present `ReScue`: a more precise gray-box approach which leverages a genetic algorithm to efficiently generate input strings.

2.3 Regexes Basics

We now define the regexes that we use for the rest of the paper. Let $\Sigma = \{ a_1, a_2, \ldots, a_n \}$ be a finite set of symbols. A *word* is an element of Σ^*, while a *language* is a set of words. We denote the empty word as ε and the concatenation of two languages L_1, L_2 as $L_1 L_2$. Let $a \in \Sigma$.

$$\mathcal{R} \in \mathbb{R} \qquad \text{Regexes}$$
$$\mathcal{R} \coloneqq \varepsilon \mid a \mid \mathcal{R}_1|\mathcal{R}_2 \mid \mathcal{R}_1{\cdot}\mathcal{R}_2 \text{ (or } \mathcal{R}_1\mathcal{R}_2) \mid \mathcal{R}_1^*$$

We assume that regexes automatically remove ε in the concatenation (this is known as a *smart-constructor* [22]), so that $\mathcal{R}{\cdot}\varepsilon$ and $\varepsilon{\cdot}\mathcal{R}$ are always simplified to $\mathcal{R}$. This allows representing regexes as they are implemented in programming languages, where ε cannot be inserted by the user in the concatenation. We define two functions to deconstruct the concatenation of a regex $\mathcal{R}$.

$$\text{head}(\mathcal{R}) \triangleq \begin{cases} \text{head}(\mathcal{R}_1) & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \\ \mathcal{R} & \text{otherwise} \end{cases} \qquad \text{tail}(\mathcal{R}) \triangleq \begin{cases} \text{tail}(\mathcal{R}_1){\cdot}\mathcal{R}_2 & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \\ \varepsilon & \text{otherwise} \end{cases}$$

Observe that since we assume that the concatenation simplifies ε, if $\text{head}(\mathcal{R}) = \varepsilon$, then $\text{tail}(\mathcal{R}) = \varepsilon$. We extend the regexes with the possibility to recognize the *empty language*, namely the empty set of words, as follows.

$$\mathcal{R} \in \mathbb{R}^{\perp} \qquad \text{Empty Regexes}$$
$$\mathcal{R} \coloneqq \varepsilon \mid a \mid \mathcal{R}_1|\mathcal{R}_2 \mid \mathcal{R}_1{\cdot}\mathcal{R}_2 \mid \mathcal{R}_1^* \mid \perp$$

Observe that $\mathbb{R} \subset \mathbb{R}^{\perp}$. Let $a \in \Sigma$. The *language recognized by a regex* $\mathcal{R} \in \mathbb{R}^{\perp}$ is defined as follows.

$$\mathcal{L}(\perp) \triangleq \varnothing \qquad \mathcal{L}(a) \triangleq \{a\} \qquad \mathcal{L}(\mathcal{R}_1\mathcal{R}_2) \triangleq \mathcal{L}(\mathcal{R}_1)\mathcal{L}(\mathcal{R}_2)$$
$$\mathcal{L}(\varepsilon) \triangleq \{\varepsilon\} \qquad \mathcal{L}(\mathcal{R}_1|\mathcal{R}_2) \triangleq \mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2) \qquad \mathcal{L}(\mathcal{R}_1^*) \triangleq \bigcup_{i \geqslant 0} \mathcal{L}(\mathcal{R}_1)^i$$

If $\mathcal{L}(\mathcal{R}_1) = \mathcal{L}(\mathcal{R}_2)$ we write $\mathcal{R}_1 =_{\mathcal{L}} \mathcal{R}_2$. Furthermore, the *union*, *intersection* and *complement* operations on regexes have respectively type $\mathbb{R}^{\perp} \times \mathbb{R}^{\perp} \to \mathbb{R}^{\perp}$, $\mathbb{R}^{\perp} \times \mathbb{R}^{\perp} \to \mathbb{R}^{\perp}$ and $\mathbb{R}^{\perp} \to \mathbb{R}^{\perp}$. We denote them by $\mathcal{R}_1 \cup^r \mathcal{R}_2$, $\mathcal{R}_1 \cap^r \mathcal{R}_2$ and $\overline{\mathcal{R}_1}^r$. Observe that if $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}$, then $\mathcal{R}_1 \cup^r \mathcal{R}_2 \in \mathbb{R}$.

Algorithm 1: Matching algorithm pseudocode.

```
1  function Match (ℛ : ℝ, w : Σ*, C : ℘(ℝ)) → bool
2      if ℛ ∈ C then
3          return false
4      switch ⟨head(ℛ), tail(ℛ)⟩ do
5          case ⟨ε, ε⟩ do
6              return w = ε
7          case ⟨a, ℛ1⟩ do
8              if w = aw1 then return Match(ℛ1, w1, ∅)
9              else return false
10         case ⟨ℛ1|ℛ2, ℛ3⟩ do
11             return Match(ℛ1ℛ3, w, C) ∨ Match(ℛ2ℛ3, w, C)
12         case ⟨ℛ1*, ℛ2⟩ do
13             return Match(ℛ1ℛ1*ℛ2, w, C ∪ {ℛ1*ℛ2}) ∨ Match(ℛ2, w, C)
```

2.4 Regex Matching

In this section, we provide the pseudocode of the matching procedure. While it is simple and concise, it models the concrete behaviour of realistic matching engines. The pseudocode ignores details specific to a particular implementation, giving a high-level description of the procedure. Our algorithm is a trivial adaptation of the one presented in [7], which models Java's matching engine. Classic textbooks about regexes confirm that matching engines in standard libraries employ a trivial backtracking procedure for the matching [15,21].

In Algorithm 1, we present the matching procedure. The logic operators are short-circuit: as soon as the input word is matched, the unexplored branches of the regex are not considered. The behaviour of function Match depends on the first constructor in the concatenation of the regex, and the remaining portion can possibly be ε. The algorithm is rather trivial, but it models two important aspects of matching engines. First, it implements a *prioritization mechanism* that: (1) tries to expand the left branch before the right branch in alternatives; (2) tries to match as many characters as possible in the body of the stars. Second, the algorithm prevents infinite ε-matching loops. Consider $(\varepsilon|a)^*$: if we remove line 3, the procedure keeps expanding the body of the star forever, never consuming any character of the input string. To prevent this, when a star is expanded, it is inserted in C, that is the set of stars that cannot be expanded again. Initially, C must be instantiated to the empty set. The stars are removed from C only when at least one character is matched. Observe that usually in matching engines the match is successful even if just a prefix of the word matches the regex: we can model this behaviour by appending Σ^* at the end of regexes.

3 Semantics

In this section, we first define a small-step operational semantics as a transition relation between the configurations of the matching engine. We then use it to put

forward a tree semantics that precisely describes the steps performed during the matching. Lastly, we use the semantics to formally define ReDoS vulnerabilities.

We extend $\mathbb{R}$ to represent when a star has been expanded and not a single character has been matched yet. The syntax of a regex $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ is given by the following grammar.

$$\mathcal{R} \in \mathbb{R}^{\mathcal{T}} \qquad \text{Transitional Regexes}$$

$$\mathcal{R} \coloneqq \varepsilon \mid a \mid \mathcal{R}_1|\mathcal{R}_2 \mid \mathcal{R}_1{\cdot}\mathcal{R}_2 \mid \mathcal{R}_1^* \mid \mathcal{R}_1^{\overline{*}}$$

It differs from traditional regexes for the *closed star*, namely $\mathcal{R}^{\overline{*}}$. It is a star that cannot be expanded again in order to prevent infinite ε-matching loops. We will formalize this concept with the transition relation. The closed stars avoid the necessity to keep a separate set of expressions (C in Algorithm 1) during the matching: the information is implicitly included in the regex.

We call a pair in $\mathbb{R}^{\mathcal{T}} \times \Sigma^* \triangleq \mathbb{S}$ a *state*, and it describes the configuration of the matching engine. The first component is the regex that the matcher is expanding, and the second is the suffix of the input word that still has to be matched. We define the function $r : \mathbb{R}^{\mathcal{T}} \to \mathbb{R}$ to transform the closed stars back into regular stars as follows.

$$r(\varepsilon) \triangleq \varepsilon \qquad r(\mathcal{R}_1|\mathcal{R}_2) \triangleq r(\mathcal{R}_1)|r(\mathcal{R}_2) \qquad r(\mathcal{R}_1^*) \triangleq r(\mathcal{R}_1)^*$$

$$r(a) \triangleq a \qquad r(\mathcal{R}_1\mathcal{R}_2) \triangleq r(\mathcal{R}_1)r(\mathcal{R}_2) \qquad r(\mathcal{R}_1^{\overline{*}}) \triangleq r(\mathcal{R}_1)^*$$

We then define the set of *actions* as $\mathbb{A} \triangleq \{ ⦶, ⦷, ⊛, ◯ \} \cup \{ ⓐ_a \mid a \in \Sigma \}$. Let $a \in \Sigma$ and $w \in \Sigma^*$. We can finally define the *transition relation* between states. It is not deterministic, but sequences of actions will be ordered later in this section.

$$\langle a, aw\rangle \xrightarrow{ⓐ_a} \langle \varepsilon, w\rangle \qquad \langle a\mathcal{R}_1, aw\rangle \xrightarrow{ⓐ_a} \langle r(\mathcal{R}_1), w\rangle$$

$$\langle \mathcal{R}_1|\mathcal{R}_2, w\rangle \xrightarrow{⦶} \langle \mathcal{R}_1, w\rangle \qquad \langle (\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, w\rangle \xrightarrow{⦶} \langle \mathcal{R}_1\mathcal{R}_3, w\rangle$$

$$\langle \mathcal{R}_1|\mathcal{R}_2, w\rangle \xrightarrow{⦷} \langle \mathcal{R}_2, w\rangle \qquad \langle (\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, w\rangle \xrightarrow{⦷} \langle \mathcal{R}_2\mathcal{R}_3, w\rangle$$

$$\langle \mathcal{R}_1^*, w\rangle \xrightarrow{⊛} \langle \mathcal{R}_1\mathcal{R}_1^{\overline{*}}, w\rangle \qquad \langle \mathcal{R}_1^*\mathcal{R}_2, w\rangle \xrightarrow{⊛} \langle \mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, w\rangle$$

$$\langle \mathcal{R}_1^*, w\rangle \xrightarrow{◯} \langle \varepsilon, w\rangle \qquad \langle \mathcal{R}_1^*\mathcal{R}_2, w\rangle \xrightarrow{◯} \langle \mathcal{R}_2, w\rangle$$

The transition relation describes all possible choices of the matching engine according to the state. Observe that with the $⊛$ action the star becomes $\overline{*}$, and it cannot be expanded again until a character is matched. In fact, the transition relation is not defined for $\mathcal{R}^{\overline{*}}$. After consuming a character of the input word, we apply the function r to mark all stars as expandable.

We now leverage the transition relation to define a tree semantics for the matching procedure. Figure 2.(a)to (d) represent the steps to obtain the semantic matching tree that we define in this section for the initial state $\langle a^*, a\rangle$. We begin by defining the set of *execution traces* for $\langle \mathcal{R}_0, w_0\rangle \in \mathbb{S}$.

$$\mathcal{T}(\langle \mathcal{R}_0, w_0\rangle) \triangleq \{ \langle \mathcal{R}_0, w_0\rangle \xrightarrow{\mathcal{A}_1} \langle \mathcal{R}_1, w_1\rangle \xrightarrow{\mathcal{A}_2} \cdots \xrightarrow{\mathcal{A}_n} \langle \mathcal{R}_n, w_n\rangle \mid \forall i \in [0, n-1] : \mathcal{A}_i \in \mathbb{A} \text{ and } \langle \mathcal{R}_i, w_i\rangle \xrightarrow{\mathcal{A}_{i+1}} \langle \mathcal{R}_{i+1}, w_{i+1}\rangle \}$$

$$\{\, \langle a^*, a\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\bigcirc} \langle \varepsilon, a\rangle \,\}$$

(a) $\mathcal{T}(\langle a^*, a\rangle)$

$$\{\, \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\bigcirc} \langle \varepsilon, a\rangle \,\}$$

(b) $\mathcal{T}_c(\langle a^*, a\rangle)$

$$\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\bigcirc} \langle \varepsilon, a\rangle$$

(c) $(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)(\langle a^*, a\rangle)$

$$\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\ \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\bigcirc\hspace{-0.8em}\triangle_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle$$

(d) $(\mathcal{F}_{\varepsilon} \circ \mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)(\langle a^*, a\rangle)$

Fig. 2

We denote the last state of a trace t as $\ell(t)$ and we define the set of *complete execution traces* as $\mathcal{T}_c(\langle \mathcal{R}, w\rangle) \triangleq \{\, t \in \mathcal{T}(\langle \mathcal{R}, w\rangle) \mid \ell(t) \not\rightarrow \}$. Observe that $\mathcal{T}_c(\langle \mathcal{R}, w\rangle)$ represents all possible executions of the matching engine from $\langle \mathcal{R}, w\rangle$ up to a state in which it is not possible to continue. We say that two traces are part of the same matching run if they have the same initial state. To build the matching tree, we need to order the traces from the first that will be explored to the last. Let t_1, t_2 be two complete execution traces in the same matching run, and let $\langle \mathcal{R}_1, w_1\rangle$ be the last state in the longest common prefix between t_1 and t_2. We impose a lexical order $\sqsubseteq$ such that $t_1 \sqsubseteq t_2$ iff the action chosen by t_1 after $\langle \mathcal{R}_1, w_1\rangle$ is either $\oplus$ or $\circledast$. Let T be a set of complete execution traces such that all of them are part of the same matching run. We denote with $\mathcal{O}_{\sqsubseteq}(T)$ the sequence of traces in T ordered by $\sqsubseteq$.

Observe that $(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)(\langle \mathcal{R}, w\rangle)$ corresponds to the ordered sequence of *all* complete execution traces. During the concrete execution, some of them will never be explored, because as soon as the state $\langle \varepsilon, \varepsilon\rangle$ is found, the procedure terminates. We want to remove from $(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)(\langle \mathcal{R}, w\rangle)$ those traces that appear after $\langle \varepsilon, \varepsilon\rangle$. Let $S = t_1, t_2, \ldots, t_n$ be a sequence of complete execution traces. We denote by $\mathcal{F}_{\varepsilon}(S)$ the sequence $t_1, t_2, \ldots, t_k$ such that t_k is the first trace for which it holds that $\ell(t_k) = \langle \varepsilon, \varepsilon\rangle$. If there is no such trace, then $k = n$ (i.e., there is an exhaustive exploration of all traces before failing).

Let S be a sequence of complete execution traces such that all of them are part of the same matching run. We denote by $\curlyvee(S)$ the tree obtained by merging the common prefixes in S.

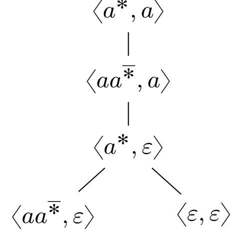

Fig. 3: Representation of $[\![a^*]\!](a)$

Definition 1 (Matching Tree Semantics). *Let* $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ *and* $w \in \Sigma^*$*. The* matching tree semantics *of* $\mathcal{R}$ *with respect to* w *is given by the following tree.*

$$[\![\mathcal{R}]\!](w) \triangleq (\curlyvee \circ \mathcal{F}_\varepsilon \circ \mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)(\langle \mathcal{R}, w\rangle)$$

Figure 3 represents $[\![a^*]\!](a)$. One can reconstruct the steps carried out by the matching engine by doing a depth-first left-to-right traversal of the semantic tree. We denote the number of nodes in a tree t with $|t|$ and its set of leaves as $\text{lvs}(t)$. We define the *language recognized by* $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ as $\mathcal{L}(\mathcal{R}) \triangleq \{ w \in \Sigma^* \mid \langle \varepsilon, \varepsilon\rangle \in \text{lvs}([\![\mathcal{R}]\!](w)) \}$. We now give the definition of ReDoS vulnerability, using the one that firstly appeared in [32], but adapted to our semantics.

Definition 2 (ReDoS Vulnerability). *Let* $\mathcal{R} \in \mathbb{R}$ *and* $n \in \mathbb{N}$*. We define* $M_{\mathcal{R}}(n) \triangleq \max\{ |[\![\mathcal{R}]\!](w)| \mid w \in \Sigma^*, |w| \leqslant n \}$*. We say that* $\mathcal{R}$ *has a* ReDoS vulnerability *iff* $M_{\mathcal{R}} \in \Omega(2^n)$*.*

4 Detection of ReDoS Vulnerabilities

In this section, we describe a framework to statically detect exponential ReDoS vulnerabilities. The analysis we propose derives from a regex an overapproximation of the set of dangerous words, namely those that can possibly cause an exponential ReDoS attack. The analysis is *sound* but not *complete*: any true vulnerability will be reported, but the algorithm can occasionally raise *false positives* (i.e., harmless regexes can be considered dangerous). Nevertheless, as discussed in Section 5, our experiments show that in practice our approach is precise and reports only 50 false positives over 74,670 regexes.

Intuitively, there is an exponential ReDoS vulnerability in a star if it is possible to match a word with at least two different traces. Consider $(a|a)^*$: a is matched in two traces by expanding the left or the right branch of the alternative. This implies that there are four traces to match aa, eight for aaa and in general 2^n for a^n. Nevertheless, $[\![(a|a)^*]\!](a^n)$ is not an exponential tree, because the match succeeds after expanding the left branch of the alternative n times. By appending a character that makes the match fail after a^n, an attacker can force the matching engine to explore all traces, effectively performing a ReDoS attack. This is the reason why $|[\![(a|a)^*]\!](a^n b)| = \Theta(2^n)$.

Algorithm 2: Regex representation of $\mathcal{M}_2$.

```
1  function M2(R : ℝ) → ℝ^⊥
2      return M2-rec(R, ∅)
3  function M2-rec(R : ℝ^T, E : ℘(ℝ^T)) → ℝ^⊥
4      if R ∈ E then
5          return ⊥
6      switch ⟨head(R), tail(R)⟩ do
7          case ⟨ε, ε⟩ ∨ ⟨R̄1*, R2⟩ do
8              return ⊥
9          case ⟨a, R1⟩ do
10             return a· M2-rec(r(R1), E)
11         case ⟨R1|R2, R3⟩ do
12             inter ← R1R3 ∩^r_∉ε R2R3
13             l ← M2-rec(R1R3, E)
14             r ← M2-rec(R2R3, E)
15             return inter ∪^r l ∪^r r
16         case ⟨R1*, R2⟩ do
17             inter ← R1R̄1*R2 ∩^r_∉ε R2
18             l ← M2-rec(R1R̄1*R2, E ∪ {R})
19             r ← R1*· M2-rec(R2, E)
20             return inter ∪^r l ∪^r r
```

First, we define a function $\mathcal{M}_2$ to extract the set of words that are matched in at least two traces in a regex $\mathcal{R}$.

$$\mathcal{M}_2(\mathcal{R}) \triangleq \{\, w \in \Sigma^+ \mid \exists t_1, t_2 \in \mathcal{T}_c(\langle\mathcal{R}, w\rangle) : t_1 \neq t_2 \text{ and } \ell(t_1) = \ell(t_2) = \langle\varepsilon, \varepsilon\rangle \,\}$$

In the analysis, we use $\mathcal{M}_2$, and since it is a possibly infinite language we need an algorithm to compute a finite representation of it. The function M2 in Algorithm 2 returns a regular expression $\mathcal{R}_1 \in \mathbb{R}^\perp$ such that $\mathcal{L}(\mathcal{R}_1) = \mathcal{M}_2(\mathcal{R})$. The correctness is proved in Appendix A. In Algorithm 2, we compute the intersection of two regexes $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^\mathcal{T}$ that does not include ε, and we denote it by $\mathcal{R}_1 \cap^r_{\not\varepsilon} \mathcal{R}_2$. In Appendix B we give a trivial procedure to compute this intersection.

The intuition behind M2 is that a word is matched in two different traces if the two branches of a choice[1] recognize some common words, that is, they have a nonempty intersection. Algorithm 2 recursively explores all regexes that can be reached from the initial one with the transition relation. When it encounters a choice, it returns the intersection of the two possible branches: the words in it are those that are matched in two different traces. Observe that since the words in $\mathcal{M}_2(\mathcal{R})$ are nonempty, we compute the intersections with $\cap^r_{\not\varepsilon}$.

To ensure termination, we keep track of which stars have already been expanded with the parameter E. When a regex in which the first construct is a star

[1] By choice we mean taking the left/right branch of an alternative or expanding/not expanding a star.

is encountered for the second time, the function returns $\bot$. This guarantees that any star will be expanded exactly once. Observe that the closed stars and the parameter E serve different purposes: the first guarantees termination during the *concrete execution* to avoid infinite ε-matching loops; the second guarantees termination of the M2-rec function.

Example 1. Consider M2($(a|a)^*$), that initially invokes M2-rec($(a|a)^*, \varnothing$). First, $(a|a)(a|a)^{\overline{*}} \cap^r_{\notin} \varepsilon =_{\mathcal{L}} \bot$ is returned; then, the recursive call M2-rec($\varepsilon, \varnothing$) immediately terminates and returns $\bot$ as well. The most interesting recursive call is M2-rec($(a|a)(a|a)^{\overline{*}}, \{(a|a)^*\}$), where the first construct in the concatenation is an alternative. The function computes and returns the nonempty intersection $a(a|a)^{\overline{*}} \cap^r_{\notin} a(a|a)^{\overline{*}} =_{\mathcal{L}} a^+$. Next, the algorithm invokes M2-rec($a(a|a)^{\overline{*}}, \{(a|a)^*\}$), which then calls M2-rec($r((a|a)^{\overline{*}}), \{(a|a)^*\}$). Since $r((a|a)^{\overline{*}}) = (a|a)^*$ and $(a|a)^*$ is in E, the algorithm terminates at line 5. To summarize, M2($(a|a)^*$) recognizes the language a^+, which is exactly $\mathcal{M}_2((a|a)^*)$.

To understand how we take advantage of M2, consider a regex $\mathcal{R}^*$ such that $\text{M2}(\mathcal{R}^*) \neq_{\mathcal{L}} \varnothing$. In this case, the set of words that are matched with at least two traces in $\mathcal{R}^*$ is not empty. Let $w \in \mathcal{L}(\text{M2}(\mathcal{R}^*))$. Since from $\mathcal{R}^*$ there are two traces to match w, then there are four traces to match w^2, eight for w^3, and in general 2^n for w^n. Furthermore, for all $n \geqslant 1$, $w^n \in \mathcal{L}(\text{M2}(\mathcal{R}^*))$. This implies that the words in M2($\mathcal{R}^*$) are possibly matched in an exponential number of traces. To have an exponential matching tree, all of them must be explored. Let $\mathcal{S} \in \mathbb{R}$, and consider the case in which w^n is matched with $\mathcal{R}^*\mathcal{S}$. By concatenating w^n with a suffix s that causes the match to fail, it is possible to force the procedure to exhaustively explore all traces, effectively resulting in an exponential matching tree. The language of suffixes that make the match fail is the language of words not accepted by $\mathcal{R}^*\mathcal{S}$, namely $\overline{\mathcal{R}^*\mathcal{S}}^r$. This is the key insight of our analysis, namely that $\text{M2}(\mathcal{R}^*)\cdot\overline{\mathcal{R}^*\mathcal{S}}^r$ accepts an overapproximation of the language of words dangerous for $\mathcal{R}^*\mathcal{S}$ that can cause exponential matching in $\mathcal{R}^*$.

With this intuition, we define the analysis $\mathcal{E} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{\bot}$ such that $\mathcal{E}(\mathcal{R}, \mathcal{P}, \mathcal{S})$ recognizes an overapproximation of the set of words dangerous for the regex $\mathcal{P}\cdot\mathcal{R}\cdot\mathcal{S}$ that can cause exponential matching in $\mathcal{R}$.

$$\mathcal{E}(\mathcal{R}, \mathcal{P}, \mathcal{S}) \triangleq \begin{cases} \bot & \text{if } \mathcal{R} = \varepsilon \text{ or } \mathcal{R} = a \\ \mathcal{E}(\mathcal{R}_1, \mathcal{P}, \mathcal{S}) \cup^r \mathcal{E}(\mathcal{R}_2, \mathcal{P}, \mathcal{S}) & \text{if } \mathcal{R} = \mathcal{R}_1|\mathcal{R}_2 \\ \mathcal{E}(\mathcal{R}_1, \mathcal{P}, \mathcal{R}_2\cdot\mathcal{S}) \cup^r \mathcal{E}(\mathcal{R}_2, \mathcal{P}\cdot\mathcal{R}_1, \mathcal{S}) & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \\ \mathcal{P}\cdot\mathcal{R}_1^*\cdot \text{M2}(\mathcal{R}_1^*)\cdot\overline{\mathcal{R}_1^*\cdot\mathcal{S}}^r \cup^r \mathcal{E}(\mathcal{R}_1, \mathcal{P}\cdot\mathcal{R}_1^*, \mathcal{R}_1^*\cdot\mathcal{S}) & \text{if } \mathcal{R} = \mathcal{R}_1^* \end{cases}$$

Initially, the analysis must be invoked as $\mathcal{E}(\mathcal{R}, \varepsilon, \varepsilon)$. It recursively explores $\mathcal{R}$, accumulating the prefixes and the suffixes of the portion that it is considering in $\mathcal{P}$ and $\mathcal{S}$. When $\mathcal{E}$ encounters a star, in addition to calling $\mathcal{E}$ recursively on the regex under the star, it also returns $\mathcal{P}\cdot\mathcal{R}_1^*\cdot \text{M2}(\mathcal{R}_1^*)\cdot\overline{\mathcal{R}_1^*\mathcal{S}}^r$. As discussed previously, $\text{M2}(\mathcal{R}_1^*)\overline{\mathcal{R}_1^*\mathcal{S}}^r$ recognizes an overapproximation of the language of words dangerous for $\mathcal{R}_1^*\mathcal{S}$ that can cause exponential matching in $\mathcal{R}_1^*$. The first construct $\mathcal{P}\cdot\mathcal{R}_1^*$ in the expression accepts the language of words that the analysis

determined to be a prefix of $\mathcal{R}_1^*\mathcal{S}$. Later in this section, we prove that the words in $\mathcal{E}(\mathcal{R}, \varepsilon, \varepsilon)$ are a sound overapproximation of the words that are dangerous for $\mathcal{R}$, and we also provide an example where the analysis loses precision.

We can perform an emptiness check on $\mathcal{E}(\mathcal{R}, \varepsilon, \varepsilon)$ to determine if there are dangerous words. If the language is empty, then $\mathcal{R}$ is not vulnerable; otherwise, we have a sound overapproximation of the words that can lead to ReDoS attacks.

Example 2. Consider $\mathcal{E}((a|a)^*, \varepsilon, \varepsilon)$.

$$\begin{aligned}
\mathcal{E}((a|a)^*, \varepsilon, \varepsilon) = &\quad (a|a)^* \cdot \mathrm{M2}((a|a)^*) \cdot \overline{(a|a)^*}^r \cup^r \mathcal{E}(a|a, (a|a)^*, (a|a)^*) \\
=_{\mathcal{L}} &\ (a|a)^* a^+ \overline{(a|a)^*}^r \cup^r \bot \\
=_{\mathcal{L}} &\ a^+ \cdot \overline{a^*}^r
\end{aligned}$$

In this case, the analysis determined that $(a|a)^*$ is vulnerable to arbitrary large sequences of as that are followed by any nonempty word not composed of as only. Observe that, effectively, $|[\![(a|a)^*]\!](a^n b)| = \Theta(2^n)$.

The following soundness theorem provides a strong guarantee that if the analysis of $\mathcal{R}$ returns an empty regex, then the size of any matching tree is at most polynomial in the length of the input word. The proof is given in Appendix C.

Theorem 1 (Soundness). *Let* $\mathcal{R} \in \mathbb{R}$.

$$\mathcal{E}(\mathcal{R}, \varepsilon, \varepsilon) =_{\mathcal{L}} \bot \implies \exists k \in \mathbb{N} : \forall w \in \Sigma^* : |[\![\mathcal{R}]\!](w)| = O(|w|^k)$$

Some patterns in regexes can cause a loss of precision in the analysis. Consider as example $\Sigma^*|(a|a)^*$ and observe how the matching procedure never explores the right (dangerous) branch of the outermost alternative. However, since the analysis does not consider the order in which the branches are explored (they are merged with $\cup^r$), $\mathcal{E}(\Sigma^*|(a|a)^*, \varepsilon, \varepsilon)$ returns the language $a^+\overline{a^*}^r$. While our analysis is not complete, our experiments show that over 74,670 regexes taken from real-world use cases, this happens only in 50 instances. This shows that patterns that can make our analysis lose precision rarely occur in practice.

The fact that the analysis returns the language of dangerous words can be useful in different scenarios. For example, it is possible to use our algorithm in a matching engine that tries to match a word only if it is not in the attack language of the input regex. The analysis we put forward can also be integrated with a static analyzer for high-level programming languages: by paring our framework with a sound string analysis, it should be possible to prove the absence of ReDoS vulnerabilities in real-world applications. This is left as future work.

Observe that even though we do not directly support lookaround assertions, it is possible to run the analysis multiple times on each assertion in a regex. In fact, if none of them is dangerous (i.e., they have empty attack languages), then the initial regex is safe. We also believe that it is possible to automatically overapproximate regexes with backreferences in a sound way (for instance, substituting `(a)*\1` with `a*a*`) to analyze them with our framework, and we would like to explore such extensions in future work.

Table 1: Attributes of the detectors.

	Type	**Sound**	**Complete**	**Language**	**Deterministic**
`rat`	*static*	✓	✗	✓	✓
`ReScue` [28]	*dynamic*	✗	✓	✗	✗
`rexploiter` [33]	*static*	✗	✗	✓	✓
`rsa` [32]	*static*	✓	✗	✗	✓
`rsa-full` [32]	*static*	✓	✓	✗	✓
`rxxr2` [27]	*static*	✗	✗	✗	✓

5 Experimental Evaluation

To assess the usefulness of the analysis we put forward, we implemented it in the `rat` [24] tool (**R**eDoS **A**bstract **T**ester, which is publicly available on Github) in less than 5000 lines of OCaml code, and we compared it to four other detectors. In our experiments, we wanted to evaluate how `rat` behaves in terms of precision and performance compared to others. We ran our experiments on a server with 128GB of RAM, with 48 Intel Xeon CPUs E5-2650 v4 @ 2.20GHz and Ubuntu 18.04.5 LTS. We considered the dataset used in [28], composed of: (1) 2,992 patterns from the Regexlib platform [2]; (2) 12,499 patterns from the Snort platform [3]; (3) 13,597 patterns extracted from 3,898 Python projects on Github in [9]. To them, we added 63,352 regexes extracted from modules in the `pypi` package manager [1] by Davis et al. [14]. From the dataset, we removed the regexes that were not properly sanitized (e.g., that contained non-printable characters) and we removed duplicates, obtaining 74,670 regexes. To the best of our knowledge, it is the first time that such a large dataset of regexes taken from real-world programs is used to compare the precision and performance of ReDoS-detection tools.

In what follows, we say that a detector is *sound* if it identifies as vulnerable all the truly vulnerable regexes, and we say that it is *complete* if all the regexes it identifies as vulnerable are truly vulnerable. Sound detectors forbid *false negatives* and complete detectors forbid *false positives*. The tools we compared `rat` to are `ReScue` [28], `rexploiter` [33], `rsa` [32] and `rxxr2` [27]. In particular, `rsa` allows the user to improve the precision of the analysis (at the cost of sacrificing some performance) with the "full" mode, that makes it the only sound and complete tool. The only dynamic detector we compare to is `ReScue`, that due to its nature never raises false positives. On the other hand, since it relies on a genetic algorithm that generates the input strings with random mutations, the analysis is not deterministic. In Section 6, we discuss the details of each approach, and in Table 1 we summarize the characteristics of the tools. While attributes reported in Table 1 summarize the expected behaviour, we found that in practice some detectors do not match the underlying theoretical results. If a detector can extract the language of dangerous words (opposed to a single exploit string) we mark the **Language** column with ✓.

Table 2: Evaluation results.

	OK	FP	FN	OOT	RTE	SKIP	TIME
rat	67,049	50	0	181	0	7,390	1:58:29
rxxr2	60,792	94	7	11	0	13,766	0:09:37
ReScue	33,541	0	43	32,200	0	8,886	325:54:19
rsa	57,243	190	1	817	242	16,177	19:58:32
rsa-full	54,823	134	1	3,174	399	16,139	39:11:21
rexploiter	53,929	31	180	327	0	20,203	9:42:47

Precision Comparison. We use the evaluation technique used in [28], which, to the best of our knowledge, is the only article that compares the precision of ReDoS detectors. We analyze each regex with the detectors setting an individual timeout of 30 seconds, and then we compare the results. If any tool can craft an exploit string of length lesser or equal to 128 characters that makes the Java 8 matching engine perform more than 10^{10} matching steps, we consider the regex to be vulnerable. During our tests, we observed that for the specific matching engine we consider, for strings of length at most 128 characters, 10^{10} matching steps are a sound threshold to clearly distinguish between exponential and non-exponential matching. We cross-reference the results of five different tools (some of which are, at least theoretically, sound) by concretely testing exploit strings on a real-world matching engine, so that we infer with high confidence the number of false positives and false negatives. We classified as vulnerable 313 regexes.

In Table 2, we report the results. The columns correspond to: number of correctly classified regexes (**OK**); false positives (**FN**); false negatives (**FN**); out of time (**OOT**); runtime errors (**RTE**); skipped (**SKIP**) (i.e., not parsed); total runtime displayed as H:MM:SS (**TIME**).

Compared to other static analyzers, `rat` reports a relatively low number of false positives: 50 over the 67,280 regexes that it parses. The only static analyzer that reports fewer false positives than `rat` is `rexploiter`, that on the other hand reports 180 false negatives and skips 20,203 regexes. Interestingly, we observed that in practice `rat` *is the only detector that does not report false negatives*. This matches our theoretical results, and it gives empirical evidence that our framework performs a sound analysis. We also observe that `rat` is the detector that parses the highest number of regexes: even more than `ReScue`, which indeed supports advanced features. This is due to the fact that `ReScue` does not support some regular patterns such as *named capturing groups* with the syntax (?P<name>pattern), that indeed `rat` can analyze.

Performance Comparison. In case a detector runs out of time for a few regexes, the total runtime in Table 2 grows sharply, not representing precisely the average performance of the tool. For this reason, we use *survival plots* to compare more faithfully the performance of the detectors. On such plot, the y-axis represents the time in milliseconds, and the x-axis is the number of regexes such that each one can be analyzed under the specified time, while the remaining

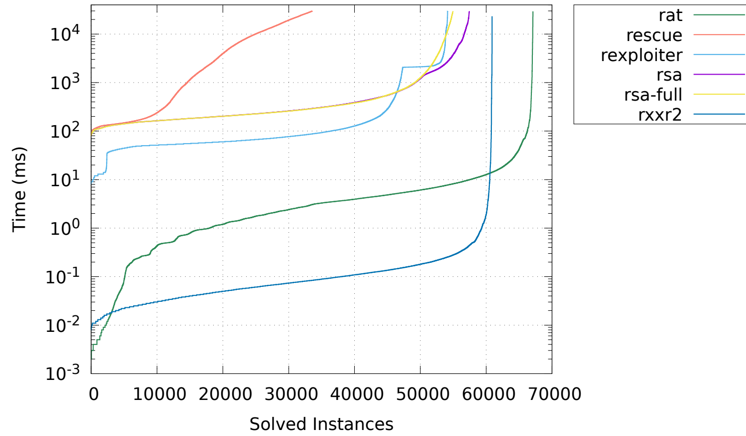

Fig. 4: Survival plot with a logarithmic y axis and linear x axis.

regexes either take longer to analyze or cannot be analyzed by the corresponding detector. No plot for x-axis and detector d means that for $74,670-x$ regexes d did not successfully complete the analysis (i.e., it either ran out of time or it had a parse/runtime error). The plot highlights the relative performance of each tool and how many regexes can be individually analyzed under a time threshold. The survival plot of our experiments is depicted in Figure 4.

Our experiments showed that `rat` is able to analyze 66,924 regexes over the 67,280 that it parses in less than one second each ($\sim$ 99.47%). As expected, `ReScue` is, due to its dynamic nature, significantly slower than static analyzers. After it, we find the cluster composed of `rsa`, `rsa-full` and `rexploiter`. Our detector is on average one to two orders of magnitude faster than them for corresponding points on abscissa x. While `rxxr2` is generally faster than `rat`, we remark that `rat` is performing a strictly more expressive analysis by returning the language of dangerous words. Furthermore, according to Table 2, `rxxr2` is not performing a sound analysis either. We also remark that `rat` analyzes 6,376 more regexes than `rxxr2`.

Discussion. We observed that in practice `rat` is one to two orders of magnitude faster than most detectors, raises a relatively low number of false positives, and it is the only analyzer that does not report false negatives. The approach based on semantic trees significantly improved the analysis' design and the easiness to reason about ReDoS vulnerabilities. It also allowed us to ignore the complexities related to transforming regexes into automata, that for some tools are sources of unsoundness and incompleteness. To the best of our knowledge, our analysis for ReDoS vulnerabilities is the first that operates directly on regexes without having to resort to automata. Regexes also make it easy to implement many performance optimizations. We integrated in `rat` two major performance improvements.

- *Character Classes Representation.* Character classes are features commonly used by programmers. For example, `\d` is a shortcut for $0|1|\dots|9$. We extend the regexes to recognize *sets of characters* instead of simple characters. With a slight adjustment to our implementation, regexes containing character classes considerably decreased their size. For example, $0|\dots|9$ has 19 constructs, while $\{0,\dots,9\}$ is a regex with a single character set construct.
- *Symbolic Operations.* In our analysis, we perform a large number of intersection and complement operations. Instead of running the algorithm to compute them, we extend again the regexes to support *symbolic intersection* and *symbolic complement*. When a complement or an intersection must be computed, we simply add its symbolic representation to the result.

6 Related Work

Wüstholz et al. [33] put forward an analysis based on automata to detect ReDoS vulnerabilities, and they implement it the `rexploiter` tool. Their approach is the closest to ours, since they can as well extract the language of dangerous words. However, the analysis is not sound nor complete, because transforming a regex into an automaton can introduce or remove vulnerabilities. For example, applying Glushkov's construction [16] to the vulnerable regex $(a^*)^*$ we obtain a non-vulnerable automaton (with respect to [33, Defn. 3]). Since they do not define an algorithm to transform regexes into automata that preserves vulnerabilities, the analysis can report both false positives and false negatives, and our experiments confirmed this.

The `rxxr2` tool is a static analyzer for exponential ReDoS vulnerabilities that infers exploit strings [27]. It is the successor of `rxxr` [18], that turned out to be unsound. Introducing a novel approach based on NFAs with prioritized transitions, `rxxr2` infers strings that can be *pumped* and lead to exponential matching. While the algorithm is sound and complete with respect to automata, transforming regexes to automata can introduce or remove vulnerabilities. Similarly to `rexploiter`, they assume that the input regex has been converted into an automaton following one of the standard constructions, so that the analysis is actually neither sound nor complete.

The framework of *prioritized NFAs* (pNFAs) [7,8] has been leveraged by Weideman et al. [32] to build the `rsa` (**R**egex**S**tatic**A**nalysis) static analyzer. The authors introduce an algorithm to translate regexes into automata that preserves the ReDoS vulnearbilities. The automata are analyzed with the framework described in [5] to determine the *degree of ambiguity* [31], which allows inferring whether there are ReDoS vulnerabilities or not. The *full* mode performs a sound and complete analysis, while the *simple* mode is only sound, but it usually runs faster. We observe that while the analysis is complete, it is strictly less expressive than ours. In fact, their framework cannot be used to extract the attack language for a regular expression, but only a finite number of exploit strings. For this reason, the two approaches are suitable for different uses: tools that need the specification of dangerous words, such as static analyzers, cannot rely

on `rsa` to extract it. Furthermore, our algorithm performs a single *emptyness check* of the attack language, while their analysis performs a *universality check* for each state of the automaton, resulting in a strictly higher complexity. Our experiments confirm that our analysis has a substantial performance advantage over the one proposed in [32].

A radically different approach to ReDoS detection is dynamic analysis. The `ReScue` tool [28] leverages a genetic algorithm to efficiently generate potentially dangerous words, that are then matched by the Java matching engine to determine if they are truly dangerous. For this reason, the tool cannot report false positives. On the other hand, there is no guarantee about the absence of false negatives. The gray-box approach makes it easy to support a wide variety of advanced features, but it has the disadvantage to be several orders of magnitude slower than static analyzers. The analysis is not deterministic, and due to its dynamic nature it is not expressive enough to compute the attack language.

Recently, many techniques have been proposed to mitigate ReDoS attacks. Cody-Kenny et al. [11] use genetic programming to substitute vulnerable regexes with safe ones. Li et al. [19] and Pan et al. [23] put forward techniques for automatic regex repair based on examples. In [13] the authors introduce a matching algorithm that leverages selective memoization to mitigate ReDoS attacks while supporting advanced regex features. Sophisticated techniques based on GPU matching [20,34] and state-merging algorithms [6] have also been proposed to speedup the matching.

7 Conclusions

In this paper, we defined a tree semantics for regular expression matching, which we leveraged to design a sound static analysis that detects ReDoS vulnerabilities. To the best of our knowledge, our ReDoS detection framework is the first one that operates directly on regexes without having to resort to automata. This allowed us to easily reason about the concrete behaviour of complex matching engines, and it opened the possibility to integrate significant performance optimizations.

We implemented our analysis in the `rat` tool, and to assess the effectiveness of our technique, we compared it to four other detectors. We found `rat` to be on average one to two orders of magnitude faster than most tools, while giving strong guarantees about the soundness of the analysis. While raising a relatively low the number of false positives, `rat` is the only ReDoS detector that did not report false negatives.

In future work, would we would like to extend our analysis to support advanced features such as backreferences and lookarounds. We believe that it is possible to automatically overapproximate those features with regular constructs in a sound way. Another interesting extension of this paper would be to integrate our framework in a static analyzer for high-level languages such as Python. We believe that by pairing `rat` with a string analysis, it is possible to prove the absence of ReDoS vulnerabilities in real-world applications.

References

1. The pypi packet manager. https://pypi.org/
2. The regexlib database. https://regexlib.com/
3. The snort database. http://www.snort.org/
4. National vulnerability database: CVE-2020-3899 (2020), https://nvd.nist.gov/vuln/detail/CVE-2020-3899
5. Allauzen, C., Mohri, M., Rastogi, A.: General algorithms for testing the ambiguity of finite automata and the double-tape ambiguity of finite-state transducers. International Journal of Foundations of Computer Science **22**(04), 883–904 (2011). https://doi.org/10.1142/s0129054111008477
6. Becchi, M., Cadambi, S.: Memory-efficient regular expression search using state merging. In: Joint Conference of the IEEE Computer and Communications Societies, INFOCOM. pp. 1064–1072 (2007). https://doi.org/10.1109/INFCOM.2007.128
7. Berglund, M., Drewes, F., van der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching **151**, 109–123 (2014). https://doi.org/10.4204/EPTCS.151.7
8. Berglund, M., van der Merwe, B.: On the semantics of regular expression parsing in the wild. Theoretical Computer Science **679**, 69–82 (2017). https://doi.org/10.1016/j.tcs.2016.09.006
9. Chapman, C., Stolee, K.T.: Exploring regular expression usage and context in Python. In: International Symposium on Software Testing and Analysis, ISSTA. pp. 282–293. ACM (2016). https://doi.org/10.1145/2931037.2931073
10. Cloudflare: Cloudflare's outage postmortem - july 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/ (2019)
11. Cody-Kenny, B., Fenton, M., Ronayne, A., Considine, E., McGuire, T., O'Neill, M.: A search for improved performance in regular expressions. In: Genetic and Evolutionary Computation Conference, GECCO. pp. 1280–1287 (2017). https://doi.org/10.1145/3071178.3071196
12. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: USENIX Security Symposium. USENIX Association (2003). https://doi.org/10.1007/11506881_10
13. Davis, J.C., Servant, F., Lee, D.: Using selective memoization to defeat regular expression denial of service (ReDoS). In: IEEE Symposium on Security and Privacy (SP). pp. 543–559. IEEE Computer Society (2021). https://doi.org/10.1109/SP40001.2021.00032
14. Davis, J.C., Coghlan, C.A., Servant, F., Lee, D.: The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE. pp. 246–256. ACM (2018). https://doi.org/10.1145/3236024.3236027
15. Friedl, J.E.F.: Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.). O'Reilly (2006), https://www.oreilly.com/library/view/mastering-regular-expressions/0596528124/
16. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys **16**(5), 1 (1961)
17. IV, L.G.M., Donohue, J., Davis, J.C., Lee, D., Servant, F.: Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In:

International Conference on Automated Software Engineering, ASE. pp. 415–426. IEEE (2019). https://doi.org/10.1109/ASE.2019.00047

18. Kirrage, J., Rathnayake, A., Thielecke, H.: Static analysis for regular expression denial-of-service attacks. In: International Conference of Network and System Security, NSS. Lecture Notes in Computer Science, vol. 7873, pp. 135–148. Springer (2013). https://doi.org/10.1007/978-3-642-38631-2_11
19. Li, Y., Xu, Z., Cao, J., Chen, H., Ge, T., Cheung, S., Zhao, H.: Flashregex: Deducing anti-redos regexes from examples. In: International Conference on Automated Software Engineering, ASE 2020. pp. 659–671 (2020). https://doi.org/10.1145/3324884.3416556
20. Lin, C., Liu, C., Chang, S.: Accelerating regular expression matching using hierarchical parallel machines on GPU. In: Global Communications Conference, GLOBECOM. pp. 1–5 (2011). https://doi.org/10.1109/GLOCOM.2011.6133663
21. López, F., Romero, V.: Mastering Python Regular Expressions. Packt Publishing Ltd (2014), https://www.packtpub.com/product/mastering-python-regular-expressions/9781783283156
22. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. Journal of Functional Programming **19**(2), 173–190 (2009). https://doi.org/10.1017/s0956796808007090
23. Pan, R., Hu, Q., Xu, G., D'Antoni, L.: Automatic repair of regular expressions. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 139:1–139:29 (2019). https://doi.org/10.1145/3360565
24. Parolini, F., Miné, A.: rat - ReDoS Abstract Tester (2022), https://github.com/parof/rat
25. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 2155–2168. ACM (2017). https://doi.org/10.1145/3133956.3134073
26. Rathnayake, A.: Semantics, analysis and security of backtracking regular expression matchers. Ph.D. thesis, University of Birmingham, UK (2015), http://etheses.bham.ac.uk/6011/
27. Rathnayake, A., Thielecke, H.: Static analysis for regular expression exponential runtime via substructural logics. CoRR **abs/1405.7058** (2014)
28. Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., Lu, J.: 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. pp. 225–235. ACM (2018). https://doi.org/10.1145/3238147.3238159
29. Stack Overflow: Stack Overflow's Outage Postmortem - July 20, 2016. https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016 (2016)
30. Staicu, C.A., Pradel, M.: 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. pp. 361–376 (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/staicu
31. Weber, A., Seidl, H.: On the degree of ambiguity of finite automata. Theoretical Computer Science **88**(2), 325–349 (1991). https://doi.org/10.1016/0304-3975(91)90381-B
32. Weideman, N., van der Merwe, B., Berglund, M., Watson, B.W.: Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In: Implementation and Application of Automata - 21st Interna-

tional Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9705, pp. 322–334. Springer (2016). https://doi.org/10.1007/978-3-319-40946-7_27
33. Wüstholz, V., Olivo, O., Heule, M.J.H., Dillig, I.: Static detection of dos vulnerabilities in programs that use regular expressions. In: Legay, A., Margaria, T. (eds.) 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. Lecture Notes in Computer Science, vol. 10206, pp. 3–20 (2017). https://doi.org/10.1007/978-3-662-54580-5_1
34. Yu, X., Becchi, M.: GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In: Computing Frontiers Conference, CF. pp. 18:1–18:10 (2013). https://doi.org/10.1145/2482767.2482791

A Proof of Correctness of M2 (See page 9)

Before proving the correctness of M2 we need some preliminary definitions. Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}, w_1, w_2 \in \Sigma^*$. If $\exists t \in \mathcal{T}(\langle \mathcal{R}_1, w_1 w_2 \rangle)$ such that $\langle \mathcal{R}_2, w_2 \rangle = \ell(t)$, then we write $\langle \mathcal{R}_1, w_1 w_2 \rangle \longrightarrow^* \langle \mathcal{R}_2, w_2 \rangle$. We need to define when a regex $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ is *valid*, namely when it is possible to obtain it by following a series of transitions from an initial regex in $\mathbb{R}$. We say that $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ is *valid* iff $\exists \mathcal{R}_1 \in \mathbb{R}, w_1, w_2 \in \Sigma^*$ such that $\langle \mathcal{R}_1, w_1 w_2 \rangle \longrightarrow^* \langle \mathcal{R}, w_2 \rangle$. Consider as example $ab^{\overline{*}}$: there is no regex in $\mathbb{R}$ that can produce a concatenated with $b^{\overline{*}}$, so that $ab^{\overline{*}}$ is not valid.

Let S be a nonempty set of regular expressions such that $\forall \mathcal{R}_1, \mathcal{R}_2 \in S$ if $\mathcal{R}_1 \neq \mathcal{R}_2$, then $|\mathcal{R}_1| \neq |\mathcal{R}_2|$ (where $|\mathcal{R}|$ is the number of constructors in the regex). We extract the longest element of S with the function $\mathrm{L} : \wp(\mathbb{R}^{\mathcal{T}}) \to \mathbb{R}^{\mathcal{T}}$ defined as $\mathrm{L}(S) \triangleq \arg\max_{\mathcal{R} \in S} |\mathcal{R}|$. Let $\mathcal{R}' \in \mathbb{R}^{\mathcal{T}}, \mathcal{R} = \mathcal{R}_1 \cdots \mathcal{R}_n \in \mathbb{R}^{\mathcal{T}}$ where for all $i \in [1 \dots n]$, $\mathcal{R}_i$ is not a concatenation. We define a function to determine whether a regex is a suffix of another modulo $\overline{*}$. $\mathrm{suff} : \mathbb{R}^{\mathcal{T}} \times \mathbb{R}^{\mathcal{T}} \to \mathrm{bool}$ is defined as $\mathrm{suff}(\mathcal{R}', \mathcal{R}_1 \cdots \mathcal{R}_n) = \mathrm{true}$ iff $\exists j \in [1 \dots n]$ such that $r(\mathcal{R}') = r(\mathcal{R}_{j+1} \dots \mathcal{R}_n)$. For example, $\mathrm{suff}(a^*a, aa^{\overline{*}}a) = \mathrm{true}$. We say that a set S is a *valid set of expansion of nested stars* if: (1) $\forall \mathcal{R} \in S, \exists \mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}$ such that $\mathcal{R} = \mathcal{R}_1^* \mathcal{R}_2$; (2) $\forall \mathcal{R}_1, \mathcal{R}_2 \in S$ such that $\mathcal{R}_1 \neq \mathcal{R}_2$ it holds $|\mathcal{R}_1| \neq |\mathcal{R}_2|$; (3) $\forall \mathcal{R} \in S \backslash \{\mathrm{L}(S)\} : \mathrm{suff}(\mathcal{R}, \mathrm{L}(S))$. An example of a valid set of expansion of nested stars is $\{ (a^*)^*, a^*(a^*)^{\overline{*}} \}$.

Then, we give the *precondition* for M2-rec$(\mathcal{R}, E)$.

1. $\mathcal{R}$ is valid;
2. E is a valid set of expansion of nested stars;
3. $\forall \mathcal{R}_i \in E$ it holds that $\mathrm{suff}(\mathcal{R}_i, \mathcal{R})$.

The second and the third conditions together imply that if $\mathcal{R} \in E$, then $\mathcal{R} = \mathrm{L}(E)$. Observe that the precondition trivially holds for M2-rec$(\mathcal{R}, \varnothing)$ if $\mathcal{R} \in \mathbb{R}$.

Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}$. If $\mathcal{R}_1 = \mathcal{R}_2$, then we define $\mathcal{M}_2^{\mathcal{R}_2} : \mathbb{R} \to \Sigma^*$ as $\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1) \triangleq \varnothing$. If $\mathcal{R}_1 \neq \mathcal{R}_2$, $\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1)$ is defined as follows.

$$\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1) \triangleq \{ w_1 w_2 \mid w_1 \in \Sigma^+, w_2 \in \Sigma^*, \exists t_1, t_2 \in \mathcal{T}(\langle \mathcal{R}_1, w_1 w_2 \rangle) : \\ t_1 \neq t_2 \wedge \ell(t_1) = \ell(t_2) = \langle \mathcal{R}_2, w_2 \rangle \wedge w_2 \in \mathcal{L}(\mathcal{R}_2) \}$$

We can now give the *postcondition* for M2-rec$(\mathcal{R}, E)$.

- If $E = \varnothing$, then $\mathcal{L}(\text{M2-rec}(\mathcal{R}, E)) = \mathcal{M}_2(\mathcal{R})$;
- If $E \neq \varnothing$, then $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}) \subseteq \mathcal{L}(\text{M2-rec}(\mathcal{R}, E)) \subseteq \mathcal{M}_2(\mathcal{R})$.

The second case in the postcondition corresponds to when the algorithm is exploring the body of a star. In this case, the function returns an overapproximation of the words that have a nonempty prefix that is matched in at least two different traces and can then reach $\mathrm{L}(E)$.

We define the set of *reachable regexes* $\text{rch} : \mathbb{R}^{\mathcal{T}} \to \wp(\mathbb{R}^{\mathcal{T}})$ as $\text{rch}(\mathcal{R}) \triangleq \{\, \mathcal{R}' \in \mathbb{R}^{\mathcal{T}} \mid \exists w_1, w_2 \in \Sigma^*, \exists t \in \mathcal{T}(\langle \mathcal{R}, w_1 w_2 \rangle) : \ell(t) = \langle \mathcal{R}', w_2 \rangle \,\}$.

Given the call M2-rec$(\mathcal{R}, E)$, we can associate to it the set of pairs $\mathcal{A}(\mathcal{R}, E)$ such that for each $\langle \mathcal{R}_1, E_1 \rangle \in \mathcal{A}(\mathcal{R}, E)$ it holds (1) M2-rec$(\mathcal{R}_1, E_1)$ is called in a subcall of M2-rec$(\mathcal{R}, E)$; (2) the control flow reaches line 6. It can be proved that for each $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}, E \in \wp(\mathbb{R}^{\mathcal{T}})$ that respect the precondition, it holds that $\langle \mathcal{R}, E \rangle \notin \mathcal{A}(\mathcal{R}, E)$, namely the configuration $\langle \mathcal{R}, E \rangle$ will never be expanded again in any subcall of M2-rec$(\mathcal{R}, E)$. This is because the algorithm keeps track, with the formal parameter E, of stars that have already been analyzed, and as soon as a regex that has a star as the first construct in the concatenation is encountered for the second time, the function terminates at line 5, never reaching line 6.

Furthermore, we observe that $\mathcal{A}(\mathcal{R}, E)$ is a finite set. This is because for each $\langle \mathcal{R}_1, E_1 \rangle \in \mathcal{A}(\mathcal{R}, E)$ it holds that $\mathcal{R}_1 \in \text{rch}(\mathcal{R})$ (since the algorithm explores all regexes that can be expanded during the concrete execution), and $\text{rch}(\mathcal{R})$ is finite. The finiteness of $\mathcal{A}(\mathcal{R}, E)$ and the fact that $\langle \mathcal{R}, E \rangle \notin \mathcal{A}(\mathcal{R}, E)$ imply the termination of the algorithm.

We can now prove by induction on $\mathcal{A}(\mathcal{R}, E)$ that the precondition implies the postcondition for M2-rec$(\mathcal{R}, E)$.

Proof. If $\mathcal{A}(\mathcal{R}, E) = \varnothing$, then there won't be any subcalls to M2-rec. There are three possible cases.

1. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R}) \rangle = \langle \varepsilon, \varepsilon \rangle$. Then, the execution reaches line 8 and $\perp$ is returned, since no word in Σ^+ is matched in two different traces from ε.
2. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R}) \rangle = \langle \mathcal{R}_1^{\overline{*}}, \mathcal{R}_2 \rangle$. Then, similarly to the previous case, the execution reaches line 8 and $\perp$ is returned, since no word can be matched if the first constructor in the concatenation is $\overline{*}$.
3. $\mathcal{R} \in E$. Then, by the second and third conditions in the precondition it must be that $\mathcal{R} = \mathrm{L}(E)$. By definition, $\mathcal{M}_2^{\mathcal{R}}(\mathcal{R}) = \varnothing$, and we conclude by observing that we correctly return $\perp$ at line 5.

If $\mathcal{A}(\mathcal{R}, E) \neq \varnothing$, then we are in the inductive case and there are subcalls to M2-rec. We consider two different scenarios: (1) $E = \varnothing$ and (2) $E \neq \varnothing$. If $E = \varnothing$ we consider again three different cases that depend on $\mathcal{R}$.

1. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R}) \rangle = \langle a, \mathcal{R}_1 \rangle$. In this case we return $a \cdot$ M2-rec$(r(\mathcal{R}_1), \varnothing)$. Since the precondition is satisfied for M2-rec$(a\mathcal{R}_1, \varnothing)$, it is satisfied also for

M2-rec$(r(\mathcal{R}_1), \varnothing)$. Furthermore, since $\langle r(\mathcal{R}_1), \varnothing\rangle \notin \mathcal{A}(r(\mathcal{R}_1), \varnothing)$, we have $\mathcal{A}(r(\mathcal{R}_1), \varnothing) \subset \mathcal{A}(a\mathcal{R}_1, \varnothing)$. We can then apply the inductive hypothesis:

$$\begin{aligned}
\mathcal{L}(a\cdot \text{M2-rec}(r(\mathcal{R}_1), \varnothing)) &= \mathcal{L}(a)\mathcal{M}_2(r(\mathcal{R}_1)) && \text{(inductive hypothesis)}\\
&= \mathcal{M}_2(a\cdot r(\mathcal{R}_1))\\
&= \mathcal{M}_2(a\mathcal{R}_1)\\
&\qquad (\forall w \in \Sigma^* : \mathcal{T}(\langle a\cdot r(\mathcal{R}_1), w\rangle) = \mathcal{T}(\langle a\mathcal{R}_1, w\rangle))
\end{aligned}$$

2. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R})\rangle = \langle \mathcal{R}_1|\mathcal{R}_2, \mathcal{R}_3\rangle$. The first action is a choice. We can divide $\mathcal{M}_2((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3)$ in three subsets: (1) those words matched by both branches of the current choice, namely $\mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3)$; (2) those words that are matched in at least two different traces after taking the left branch, namely $\mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3)$; (3) those words that are matched in at least two different traces after taking the right branch, namely $\mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3)$. Similarly to the previous case, the precondition in each subcall is satisfied. Furthermore, $\mathcal{A}(\mathcal{R}_1\mathcal{R}_3, \varnothing) \subset \mathcal{A}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, \varnothing)$ and $\mathcal{A}(\mathcal{R}_2\mathcal{R}_3, \varnothing) \subset \mathcal{A}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, \varnothing)$ hold. We can then apply the inductive hypothesis: $\mathcal{L}(\text{M2-rec}(\mathcal{R}_1\mathcal{R}_3, \varnothing)) = \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3)$ and $\mathcal{L}(\text{M2-rec}(\mathcal{R}_2\mathcal{R}_3, \varnothing)) = \mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3)$ hold. Observing that we return $(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_2, \varnothing) \cup^r \text{M2-rec}(\mathcal{R}_2\mathcal{R}_2, \varnothing)$, we can conclude:

$$\begin{aligned}
&\mathcal{L}(\text{M2-rec}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, \varnothing))\\
&= \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_3, \varnothing) \cup^r \text{M2-rec}(\mathcal{R}_2\mathcal{R}_3, \varnothing))\\
&= \mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3) \qquad \text{(inductive hypothesis)}\\
&= \mathcal{M}_2((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3)
\end{aligned}$$

3. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R})\rangle = \langle \mathcal{R}_1^*, \mathcal{R}_2\rangle$. Then the first action is a choice. Similarly to the previous case, we can divide $\mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2)$ in three subsets: (1) the words matched by both branches of the current choice (that is to expand the star or not), namely $\mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2)$; (2) the words that are matched in at least two different traces in the body of the star and that can reach $\mathcal{R}_1^*\mathcal{R}_2$, namely $\mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$; (3) the words that are matched in at least two different traces in $\mathcal{R}_2$, namely $\mathcal{M}_2(\mathcal{R}_2)$. Observe that the words in $\mathcal{M}_2(\mathcal{R}_2)$ have as prefix language all the words that can be matched in $\mathcal{R}_1^*$, so that the last set actually is $\mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)$. If the precondition holds for M2-rec$(\mathcal{R}_1^*\mathcal{R}_2, \varnothing)$, then it holds for the subcalls. Furthermore, $\mathcal{A}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \subset \mathcal{A}(\mathcal{R}_1^*\mathcal{R}_2, \varnothing)$ and $\mathcal{A}(\mathcal{R}_2, \varnothing) \subset \mathcal{A}(\mathcal{R}_1^*\mathcal{R}_2, \varnothing)$, so that by inductive hypothesis we have:

$$\mathcal{L}(\mathcal{R}_1^*\cdot \text{M2-rec}(\mathcal{R}_2, \varnothing)) = \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)$$

$$\mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2) \subseteq \mathcal{L}(\text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\})) \subseteq \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$$

Observing that we return $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2) \cup^r$ M2-rec$(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r$ $\mathcal{R}_1^*\cdot$ M2-rec$(\mathcal{R}_2, \varnothing)$, we can conclude:

$$
\begin{aligned}
&\mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2)\\
&= \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2) \cup \mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)\\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r \mathcal{R}_1^*\cdot \text{M2-rec}(\mathcal{R}_2, \varnothing))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(inductive hypothesis)}\\
&\subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(inductive hypothesis)}\\
&= \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2) \qquad\qquad (\mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2) \subseteq \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \subseteq \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2))
\end{aligned}
$$

So that $\mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2\cap^r_{\notin}\mathcal{R}_2)\cup^r$M2-rec$(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\})\cup^r\mathcal{R}_1^*\cdot$ M2-rec$(\mathcal{R}_2, \varnothing))$ equals $\mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2)$.

We then consider the other case, namely $E \neq \varnothing$. There are three cases, depending on $\mathcal{R}$.

1. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R})\rangle = \langle a, \mathcal{R}_1\rangle$. In this case we return $a\cdot$ M2-rec$(r(\mathcal{R}_1), E)$. By the fact that the precondition is satisfied for M2-rec$(\mathcal{R}, E)$, it is satisfied also for M2-rec$(r(\mathcal{R}_1), E)$. Furthermore, we have $\mathcal{A}(r(\mathcal{R}_1), E) \subset \mathcal{A}(a\mathcal{R}_1, E)$. We can then apply the inductive hypothesis and obtain:

$$
\begin{aligned}
\mathcal{M}_2^{\mathrm{L}(E)}(a\mathcal{R}_1) &= \mathcal{M}_2^{\mathrm{L}(E)}(a\cdot r(\mathcal{R}_1))\\
&\qquad\qquad (\forall w \in \Sigma^* : \mathcal{T}(\langle a\mathcal{R}_1, w\rangle) = \mathcal{T}(\langle a\cdot r(\mathcal{R}_1), w\rangle))\\
&= \mathcal{L}(a)\mathcal{M}_2^{\mathrm{L}(E)}(r(\mathcal{R}_1))\\
&\subseteq \mathcal{L}(a\cdot \text{M2-rec}(r(\mathcal{R}_1), E)) &&\text{(inductive hypothesis)}\\
&\subseteq \mathcal{L}(a)\mathcal{M}_2(r(\mathcal{R}_1)) &&\text{(inductive hypothesis)}\\
&= \mathcal{M}_2(a\cdot r(\mathcal{R}_1))\\
&= \mathcal{M}_2(a\mathcal{R}_1) \qquad (\forall w \in \Sigma^* : \mathcal{T}(\langle a\cdot r(\mathcal{R}_1), w\rangle) = \mathcal{T}(\langle a\mathcal{R}_1, w\rangle))
\end{aligned}
$$

2. $\langle \text{head}(\mathcal{R}), \text{tail}(\mathcal{R})\rangle = \langle \mathcal{R}_1|\mathcal{R}_2, \mathcal{R}_3\rangle$. The first action is a choice. We can divide $\mathcal{M}_2^{\mathrm{L}(E)}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3)$ in three subsets: (1) the words in $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1\mathcal{R}_3)$; (2) the words in $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2\mathcal{R}_3)$; (3) the words w_1w_2 with $w_1 \in \Sigma^+, w_2 \in \Sigma^*$ such that $\langle \mathcal{R}_1\mathcal{R}_3, w_1w_2\rangle \longrightarrow^* \langle \mathrm{L}(E), w_2\rangle$, $\langle \mathcal{R}_2\mathcal{R}_3, w_1w_2\rangle \longrightarrow^* \langle \mathrm{L}(E), w_2\rangle$ and $w_2 \in \mathcal{L}(\mathrm{L}(E))$. This set corresponds to those words that have a nonempty prefix that can be matched by both branches of the alternative and can reach $\mathrm{L}(E)$. Let I be this set: observe that it is a subset of $\mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3)$. The precondition in each subcall is satisfied, $\mathcal{A}(\mathcal{R}_1\mathcal{R}_3, E) \subset \mathcal{A}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, E)$ and $\mathcal{A}(\mathcal{R}_2\mathcal{R}_3, E) \subset \mathcal{A}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, E)$ hold. We can then apply the inductive hypothesis and, observing that we return $(\mathcal{R}_1\mathcal{R}_3\cap^r_{\notin}\mathcal{R}_2\mathcal{R}_3)\cup^r$M2-rec$(\mathcal{R}_1\mathcal{R}_3, E)\cup^r$

M2-rec$(\mathcal{R}_2\mathcal{R}_3, E)$, we obtain:

$$\begin{aligned}
&\mathcal{M}_2^{\mathrm{L}(E)}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3)\\
&= I \cup \mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2\mathcal{R}_3)\\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap^r_{\not\in} \mathcal{R}_2\mathcal{R}_3) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_3, E) \cup^r \text{M2-rec}(\mathcal{R}_2\mathcal{R}_3, E))\\
&\qquad\qquad (\text{inductive hypothesis and } I \subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\not\in} \mathcal{R}_2\mathcal{R}_3))\\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap^r_{\not\in} \mathcal{R}_2\mathcal{R}_3)) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3) \qquad (\text{inductive hypothesis})\\
&= \mathcal{M}_2((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3) \qquad (\text{analogous to subcase } (\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3 \text{ if } E = \varnothing)
\end{aligned}$$

3. $\langle \mathrm{head}(\mathcal{R}), \mathrm{tail}(\mathcal{R})\rangle = \langle \mathcal{R}_1^*, \mathcal{R}_2\rangle$. The first action in this case is a choice. We can divide the set $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1^*\mathcal{R}_2)$ in three subsets: (1) the words in $\mathcal{M}_2^{\mathrm{L}(E\cup\{\mathcal{R}_1^*\mathcal{R}_2\})}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$; (2) the words in $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2)$ (they have as prefix language all the words that can be matched in $\mathcal{R}_1^*$, so that actually the set corresponds to $\mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2)$); (3) the words w_1w_2 with $w_1 \in \Sigma^+, w_2 \in \Sigma^*$ such that $\langle \mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, w_1w_2\rangle \longrightarrow^* \langle \mathrm{L}(E), w_2\rangle$, $\langle \mathcal{R}_2, w_1w_2\rangle \longrightarrow^* \langle \mathrm{L}(E), w_2\rangle$ and $w_2 \in \mathcal{L}(\mathrm{L}(E))$. This set corresponds to those words that have a nonempty prefix that can be matched by both the expansion of the star and $\mathcal{R}_2$, and can then reach $\mathrm{L}(E)$. Let I be this set: observe that it is a subset of $\mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\not\in} \mathcal{R}_2)$. The precondition in each subcall is satisfied, $\mathcal{A}(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \subset \mathcal{A}(\mathcal{R}_1^*\mathcal{R}_2, E)$ and $\mathcal{A}(\mathcal{R}_2, E) \subset \mathcal{A}(\mathcal{R}_1^*\mathcal{R}_2, E)$ hold. We can then apply the inductive hypothesis and, observing that we return $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\not\in} \mathcal{R}_2) \cup^r$ M2-rec$(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r \mathcal{R}_1^* \cdot$ M2-rec$(\mathcal{R}_2, E)$, we obtain:

$$\begin{aligned}
&\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1^*\mathcal{R}_2)\\
&= I \cup \mathcal{M}_2^{\mathrm{L}(E\cup\{\mathcal{R}_1^*\mathcal{R}_2\})}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2)\\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\not\in} \mathcal{R}_2) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r \mathcal{R}_1^* \cdot \text{M2-rec}(\mathcal{R}_2, E))\\
&\qquad\qquad (\text{inductive hypothesis and } I \subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\not\in} \mathcal{R}_2))\\
&\subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\not\in} \mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_2) \qquad (\text{inductive hypothesis})\\
&= \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2) \qquad (\text{analogous to subcase } \mathcal{R} = \mathcal{R}_1^*\mathcal{R}_2 \text{ if } E = \varnothing)
\end{aligned}$$

B Algorithm to compute $\cap^r_{\not\in}$

We give function non-eps-inter in Algorithm 3 to compute the intersection of two regexes in $\mathbb{R}^{\mathcal{T}}$ without ε. The intuition is that the function non-eps-lang$(\mathcal{R})$ computes a regex $\mathcal{R}_{\not\in}$ such that $\mathcal{L}(\mathcal{R}_{\not\in}) = \mathcal{L}(\mathcal{R})\backslash\{\varepsilon\}$, and then the intersection of the two non-epsilon languages is computed using $\cap^r$.

C Proof of Theorem 1 (See page 11)

Before proving the soundness of the exponential analysis, we need some preliminary definitions and results. First, if $\forall w \in \Sigma^* : |[\![\mathcal{R}]\!](w)| = O(|w|)$ we simply

Algorithm 3: Algorithm that computes $\cap^r_{\not\varepsilon}$.

```
1  function non-eps-inter(R1 : R^T, R2 : R^T) → R^⊥
2      return non-eps-lang(R1) ∩^r non-eps-lang(R2)
3  function non-eps-lang(R : R^T) → R^⊥
4      switch ⟨head(R), tail(R)⟩ do
5          case ⟨ε, ε⟩ ∨ ⟨R1^*̄, R2⟩ do
6              return ⊥
7          case ⟨a, R1⟩ do
8              return a·(r(R1))
9          case ⟨R1|R2, R3⟩ do
10             return non-eps-lang(R1R3) ∪^r non-eps-lang(R2R3)
11         case ⟨R1^*, R2⟩ do
12             return non-eps-lang(R1R1^*̄R2) ∪^r non-eps-lang(R2)
```

write lin($\mathcal{R}$). We define the *frontier* of a state $f : \mathbb{S} \to \mathbb{S}^*$ as $f(\langle \mathcal{R}, aw\rangle) \triangleq \langle r(\mathcal{R}_1), w\rangle, \dots, \langle r(\mathcal{R}_n), w\rangle$, where $[\![a\mathcal{R}_1]\!](aw), \dots, [\![a\mathcal{R}_n]\!](aw)$ is the possibly empty ordered sequence of subtrees of $[\![\mathcal{R}]\!](aw)$ such that the next action is matching the first character a. For example, $f(\langle (a|a)^*, ab\rangle) = \langle (a|a)^*, b\rangle, \langle (a|a)^*, b\rangle$. The frontier of the empty word is $f(\langle \mathcal{R}, \varepsilon\rangle) \triangleq \varnothing$. We abuse the notation and we generalize the frontier to sequences of states: $f(\langle \mathcal{R}_1, w\rangle, \dots, \langle \mathcal{R}_n, w\rangle)$ is the ordered concatenation of $f(\langle \mathcal{R}_1, w\rangle), \dots, f(\langle \mathcal{R}_n, w\rangle)$.

Observe that the height of the matching tree has as upper bound the length of the input string. The following lemma formalizes this intuition.

Lemma 1. *Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}, w \in \Sigma^*$ and h be the height of $[\![\mathcal{R}]\!](w)$. Then, $h = O(|w|)$.*

Proof. Let m be the least integer such that $f^m(\langle \mathcal{R}, w\rangle)$ is empty. Since the number of nodes between the frontiers does not depend on the length of the input word but only on the regex, $h = \Theta(m)$. Observe that the words in the states of $f^n(\langle \mathcal{R}, w\rangle)$ have $|w| - n$ characters. By defintion of f, when $n = |w|$ the next frontier must be empty. This implies that m cannot be greater than $|w|$, so that $h = O(|w|)$.

The previous lemma implies that exponential ReDoS vulnerabilities arise when the *width* of a semantic tree can grow exponentially.

We recall from Appendix A the definition of *reachable regexes* rch : $\mathbb{R}^{\mathcal{T}} \to \wp(\mathbb{R}^{\mathcal{T}})$, defined as $\text{rch}(\mathcal{R}) \triangleq \{\, \mathcal{R}' \in \mathbb{R}^{\mathcal{T}} \mid \exists w_1, w_2 \in \Sigma^*, \exists t \in \mathcal{T}(\langle \mathcal{R}, w_1 w_2\rangle) : \ell(t) = \langle \mathcal{R}', w_2\rangle \,\}$. The following result formalizes the intuition that if there are no words that can be matched in two different traces, the match is linear in the worst case.

Lemma 2. *Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$.*

$$\mathcal{M}_2(\mathcal{R}) = \varnothing \implies \text{lin}(\mathcal{R})$$

Proof. Intuitively, if there is no word that is matched in two different traces, there is no ambiguity and the matching is linear in the worst case. More formally, we

prove that $\forall w \in \Sigma^* : |[\![\mathcal{R}]\!](w)| = O(|w|)$. Consider the portion of the semantic tree from the root $\langle \mathcal{R}, w \rangle$ to the nodes in $f(\langle \mathcal{R}, w \rangle)$. Observe that those are the only nodes that possibly have subtrees: all the others are either internal nodes to the portion that we are considering or do not have children. Observe also that the number of nodes between the root and the frontier does not depend on $|w|$, but just on $\mathcal{R}$ and the first character of w, if there is any.

We observe that the number of nodes in $f(\langle \mathcal{R}, w \rangle)$ is bounded by $|\operatorname{rch}(\mathcal{R})|$, since there is at most one occurrence of any regex $\mathcal{R}_1 \in \operatorname{rch}(\mathcal{R})$ in $f(\langle \mathcal{R}, w \rangle)$. This is because if there were two occurrences of any $\mathcal{R}_1 \in \operatorname{rch}(\mathcal{R})$, this would violate the hypothesis $\mathcal{M}_2(\mathcal{R}) = \varnothing$: there would be two different traces to match the first character of w. Furthermore, for the same reason it holds that for each $i \in [1 \dots |w|]$:

$$|f^i(\langle \mathcal{R}, w \rangle)| \leqslant |\operatorname{rch}(\mathcal{R})|$$

We also observe that the number of nodes between one frontier and the next does not depend on $|w|$. Since by Lemma 1 the height of a semantic tree is always linear in the length of the word, we conclude that $|[\![\mathcal{R}]\!](w)| = O(|w| \cdot |\operatorname{rch}(\mathcal{R})|)$.

Let $\mathcal{R} \in \mathbb{R}$, then we define $S : \mathbb{R} \to \mathbb{N}$ as

$$S(\mathcal{R}) \triangleq \begin{cases} 0 & \text{if } \mathcal{R} = a \text{ or } \mathcal{R} = \varepsilon \\ S(\mathcal{R}_1) + S(\mathcal{R}_2) & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \text{ or } \mathcal{R} = \mathcal{R}_1|\mathcal{R}_2 \\ 1 + S(\mathcal{R}_1) & \text{if } \mathcal{R} = \mathcal{R}_1^* \end{cases}$$

We can finally prove the soundness of the exponential analysis. Let $\mathcal{R} \in \mathbb{R}$.

$$\mathcal{E}(\mathcal{R}, \varepsilon, \varepsilon) =_{\mathcal{L}} \perp \implies \forall w \in \Sigma^* : |[\![\mathcal{R}]\!](w)| = O(|w|^{S(\mathcal{R})})$$

Proof. We prove the theorem by induction on $S(\mathcal{R})$. The base case is $S(\mathcal{R}) = 0$, namely in $\mathcal{R}$ there are no stars. We observe that stars are the only constructors that allow matching an arbitrary number of character, which implies that the size of each matching tree is bounded by a constant that does not depend on the input word, namely $\forall w \in \Sigma^* : |[\![\mathcal{R}]\!](w)| = O(1)$. This can be seen as a consequence that $\mathcal{L}(\mathcal{R})$ is finite.

The inductive case is $S(\mathcal{R}) \geqslant 1$. The only case that we consider is when $\operatorname{head}(\mathcal{R}) = \mathcal{R}_1^*$ and $\operatorname{tail}(\mathcal{R}) = \mathcal{R}_2$. All other cases can be reduced to this: regex constructors that are not stars can match only a constant number of character before reaching a star. Observe that by definition of $\mathcal{E}$, $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \varepsilon, \varepsilon) =_{\mathcal{L}} \perp$ implies $\mathcal{E}(\mathcal{R}_2, \mathcal{R}_1^*, \varepsilon) =_{\mathcal{L}} \perp$. Since the prefixes do not change the emptiness of the result, $\mathcal{E}(\mathcal{R}_2, \varepsilon, \varepsilon) =_{\mathcal{L}} \perp$. By observing that $S(\mathcal{R}_2) < S(\mathcal{R}_1^*\mathcal{R}_2)$, we can apply the inductive hypothesis and obtaint that $\forall w \in \Sigma^* : |[\![\mathcal{R}_2]\!](w)| = O(|w|^{S(\mathcal{R}_2)}) = O(|w|^{S(\mathcal{R}_1^*\mathcal{R}_2)-1})$. Therefore, for all $w \in \Sigma^*$, if w' is a suffix of w, all subtrees $[\![\mathcal{R}_2]\!](w')$ of $[\![\mathcal{R}]\!](w)$ have size at most superlinear in $|w'|$, which implies that the size is at most superlinear in $|w|$. Since $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \varepsilon, \varepsilon) =_{\mathcal{L}} \perp$, then $\mathrm{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \perp$. By Lemma 2 we can observe that matching any word in $\mathcal{R}_1^*$ is at most linear in the length of the input word. Let $w \in \Sigma^*$. In $[\![\mathcal{R}_1^*\mathcal{R}_2]\!](w)$ there are at most $|w|$ nodes of type $\langle \mathcal{R}_2, w' \rangle$ after matching any prefix of w in $\mathcal{R}_1^*$, namely at most one

for any prefix of w. This is because $\mathrm{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \bot$ implies that it is not possible to have two different traces that match any prefix of w. These observations imply that the matching tree can be decomposed in the part in which $\mathcal{R}_1^*$ is expanded (which is linear), and at most $|w|$ subtrees in which $\mathcal{R}_2$ is expanded. All those subtress have size $O(|w|^{S(\mathcal{R}_1^*\mathcal{R}_2)-1})$. Therefore, we obtain:

$$\begin{aligned}
|[\![\mathcal{R}_1^*\mathcal{R}_2]\!](w)| &= O(|w|) + \sum_{i=1}^{|w|} O(|w|^{S(\mathcal{R}_1^*\mathcal{R}_2)-1}) \\
&\quad (\mathrm{lin}(\mathcal{R}_1^*) \text{ and } \forall w' \text{ suffix of } w : |[\![\mathcal{R}_2]\!](w')| = O(|w|^{S(\mathcal{R}_1^*\mathcal{R}_2)-1})) \\
&= O(|w|) + |w| O(|w|^{S(\mathcal{R}_1^*\mathcal{R}_2)-1}) \\
&= O(|w|^{S(\mathcal{R}_1^*\mathcal{R}_2)})
\end{aligned}$$

This proves the theorem. Observe that $\mathcal{E}(\mathcal{R}_1^*, \varepsilon, \mathcal{R}_2) =_{\mathcal{L}} \bot$ can be caused not only by $\mathrm{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \bot$, but also by $\overline{\mathcal{R}_1^*\mathcal{R}_2}^r =_{\mathcal{L}} \bot$. The only language that has as complement the empty language is Σ^*, which implies that $\mathcal{L}(\mathcal{R}_1^*\mathcal{R}_2) = \Sigma^*$. This case is then analogous to the previous one, because even though there might be an exponential number of traces to match a word in $\mathcal{R}_1^*$, only one is actually expanded since $\mathcal{R}_1^*\mathcal{R}_2$ accepts any word. In this case, there exists no suffix that can make the match fail and trigger the exhaustive exploration of the set of traces.