# Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks (Extended Version)

Francesco Parolini, Antoine Miné

# Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks (Extended Version)

Francesco Parolini 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 74,669 regular expressions, we observed that in 99.78% of the instances the analysis terminates in less than one second. We compared `rat` to seven 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, Privacy.

# 1 Introduction

Regular expressions (regexes) are often used to verify that strings in programs match a given pattern. Modern programming languages offer support for regexes in standard libraries, and this encourages programmers to take advantage of them. However, matching engines in 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* [1].

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 [2], while in [3] the authors found that 10% of the Node.js-based web services they examined are vulnerable to ReDoS. In this already harsh scenario, in [4] 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 [5], Cloudflare [6], and iCloud [7]. Since it is difficult to detect ReDoS vulnerabilities with manual inspection, it is necessary to automate this 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 [8, 9, 10, 11], 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 [12] can hardly be used in practice, since performing dynamic testing on exponential algorithms can be excessively costly. Heuristics-based syntactical analyzer [13, 14, 15, 16] try to detect vulnerabilities by matching regex constructs against potentially dangerous patterns. However, these tools do not offer guarantees about the quality of the results, often reporting both false positives and false negatives.

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 *overapproximation* 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` [17], 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 (it does not raise false negatives) and raising only 49 false alarms over 74,669 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.

To summarize, this paper makes the following contributions:

- We introduce a novel *tree semantics* to describe the behaviour of matching engines and we leverage it to formally define ReDoS vulnerabilities;
- We put forward a sound analysis that extracts an overapproximation of the language of words that can cause an exponential ReDoS attack for a regex. Our framework does not depend on automata and allows us to soundly reason about the concrete behaviour of the matching engines;
- We implement the analysis in a tool called `rat`. We also compare the performance and the precision of `rat` to seven other detectors. In our evaluation, we find that `rat` is on average one to two orders of magnitude faster than most other approaches, while being strictly more expressive than the others. More interestingly, `rat` is the only detector that does not report false negatives.

The remainder of the paper is organized as follows. Section 2 gives an introduction to ReDoS vulnerabilities and regex basics. Section 3 introduces the *tree semantics*, which allows formalizing ReDoS vulnerabilities and reasoning about the concrete behaviour of matching engines. Section 4 describes a sound analysis to detect possible ReDoS attacks, while Section 5 compares the precision and performance of our implementation of the analysis with seven other detectors. Section 6 describes in depth the different existing approaches to ReDoS detection. Finally, Section 7 concludes the paper with a discussion of future work.

This is an extended and revised version of a conference paper that appeared in Theoretical Aspects of Software Engineering (TASE 2022) [18]. In this new version, we add to our experiments a comparison with a new class of analyzers, namely the *heuristics-based* analyzers, that do not offer theoretical guarantees about the precision of the analysis. In Section 2.3 we describe such approach, and in Section 5 we add to our experimental comparison three new heuristics-based tools. We also include the soundness proof for our analysis (Theorem 2). We use two main intermediate theoretical results, which we incorporate in the main body of the article (Lemma 1 and Lemma 2). We also formalize the correctness of our algorithm to extract the nondeterminism in regular expressions (Algorithm 2) in Theorem 1, and we prove it in A. Furthermore, we enrich the paper by adding Example 2, Figure 3b, and Algorithm 3.

# 2 Background

## 2.1 Regex Matching

The majority of modern programming languages offer support for regular expression matching in their standard library. While language membership is well-known to be computable in linear time in the length of input strings for regular languages [19, 20], matching engines designers often decide to increase the expressivity of regular expressions by introducing *backreferences* and *lookaround assertions* [21, 22], making the matching less efficient. These two features allow the user to express non-regular patterns, extending the capabilities of the matching engine. Backreferences and lookarounds are radically different from other extensions of classic regexes that do not change the expressive power (such as *character classes* [21, 22]), as they cannot be converted into regular constructs. Matching engines that support backreferences and lookarounds use *backtracking algorithms*, which have exponential worst-time complexity. Since in this work we target backtracking-based matching engines, in Section 2.5 we introduce the pseudocode for the backtracking matching procedure. As backreferences and lookarounds are, for the moment, not in the scope of our analysis, we present a simple version of the matching procedure that does not consider them.

While the majority of languages allow backreferences and lookarounds, there exist some exceptions. For instance, Rust uses well-known techniques based on finite-state machines [19, 20] to guarantee superior performance. This comes at the cost of forbidding backreferences and lookaround assertions. In Table 1 we report the two main approaches to regex matching, their complexity with respect to the

| Algorithm | Complexity | Used In |
|---|---|---|
| Finite-State Machine [19, 20] | Linear | Rust [23, 24], RE2 Engine [25] |
| Backtracking [21, 22] | Exponential | Javascript (V8 runtime) [26, 27], Java [28, 29], PHP [30, 31], Perl [32, 33] Python [34, 35], Ruby [36, 37], |

Table 1: Matching algorithms comparison

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

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

length of input strings, and some of the programming languages and matching engines that use them. Observe that there are also other approaches to regex matching, such as derivatives-based matching [38, 39, 40], but they are not widely used in matching engines.

## 2.2 ReDoS Vulnerabilities

The majority of programming languages that offer support for regexes in standard libraries are vulnerable to ReDoS attacks. 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 [41] 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.

Most programming languages employ matching engines with exponential worst-time complexity to support *backreferences* and *lookarounds* [21, 22], which are non-regular constructs. Since our analysis is limited, for now, to classic regular expressions, we, as many other analyzers, do not support such non-regular features. Nevertheless, our approach is sufficient to analyze the great majority of regexes in real-world applications: in [2] 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 10% of the web services they considered present ReDoS vulnerabilities [3]. 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.

## 2.3 ReDoS Detection

There are three main approaches to ReDoS detection:

1. *Heuristics-based static detection.* Heuristics-based static analyzers are tools that try to determine whether a regex is vulnerable or not using heuristics. They usually match the constructs of the input regex against a set of potentially dangerous patterns. For instance, `safe-regex` [14] checks that regexes do not present nested stars. By performing simple syntactic checks, these tools are typically faster than others. On the other hand, since they do not rely on formal semantics, they can report both false positives and false negatives, and they usually provide low-quality results. The tools `safe-regex` [14], `regexploit` [15], and `redos-detector` [16] are examples of heuristics-based static analyzers.

2. *Semantics-based static detection.* There are many different approaches to semantics-based static ReDoS detection [8, 42, 9, 10, 11], 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 which is also semantics-based, but operates on regexes instead of automata.

3. *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` [43], to detect ReDoS vulnerabilities,

in [12] the authors present `ReScue`: a more precise gray-box approach which leverages a genetic algorithm to efficiently generate input strings.

As described in Section 5, in our experiments we find that heuristics-based static analyzers raise a sensibly higher number of false positives and false negatives compared to other approaches. Nevertheless, heuristics-based detectors are the mostly used tools in practice. For instance, `safe-regex` [14] averages from 18,000 to 20,000 downloads per week on `npm` [44].

## 2.4 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 $\Sigma^*$, while a *language* is a set of words. We denote the empty word as $\varepsilon$ and the concatenation of two languages $L_1, L_2$ as $L_1L_2$. Let $a \in \Sigma$.

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

We assume that regexes automatically remove $\varepsilon$ in the concatenation (this is known as a *smart-constructor* [40]), 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 $\varepsilon$ cannot be inserted by the user in the concatenation. We define two functions to deconstruct the concatenation of a regex $\mathcal{R}$.

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

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

$$\begin{aligned} &\mathcal{R} \in \mathbb{R}^{\perp} && \text{(Possibly Empty Regexes)} \\ &\mathcal{R} \coloneqq \varepsilon \mid a \mid \mathcal{R}_1|\mathcal{R}_2 \mid \mathcal{R}_1{\cdot}\mathcal{R}_2 \mid \mathcal{R}_1^* \mid \perp \end{aligned}$$

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.

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

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 $\overline{\mathcal{R}_1}^r$. If $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}$, then $\mathcal{R}_1 \cup^r \mathcal{R}_2 \in \mathbb{R}$. Observe that $\mathcal{R}_1|\mathcal{R}_2$ is different from $\mathcal{R}_1 \cup^r \mathcal{R}_2$: the first is a regex constructor, while the second is a function that returns a regex that accepts the union of the languages of $\mathcal{R}_1$ and $\mathcal{R}_2$. While the union can be implemented using the alternative constructor, it is also possible to perform simplifications and optimisations on the result, such as $a \cup^r a = a$ instead of $a|a$. Our algorithms use the union without requiring a specific implementation. We define the function $s : \mathbb{R} \to \mathbb{N}$ that returns the number of stars in a regex as follows.

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

## 2.5 Backtracking 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 [28], which models Java's matching engine. Classic textbooks about regexes [21, 22] confirm that matching engines in standard libraries employ a trivial backtracking procedure for the matching. As backreferences and lookarounds are, for the moment, not in the scope of our analysis, we present a simple version of the matching procedure that does not consider them.

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 $\varepsilon$. 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 $\varepsilon$-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 the empty set. The stars are removed from $C$ only when at least one character is matched.

**Algorithm 1:** Matching algorithm pseudocode

```
function Match ($\mathcal{R} : \mathbb{R}, w : \Sigma^*, C : \wp(\mathbb{R})$) → bool
    if $\mathcal{R} \in C$ then
        return false
    switch $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle$ do
        case $\langle \varepsilon, \varepsilon \rangle$ do
            return $w = \varepsilon$
        case $\langle a, \mathcal{R}_1 \rangle$ do
            if $w = aw_1$ then return $\mathrm{Match}(\mathcal{R}_1, w_1, \varnothing)$
            else return false
        case $\langle \mathcal{R}_1 | \mathcal{R}_2, \mathcal{R}_3 \rangle$ do
            return $\mathrm{Match}(\mathcal{R}_1\mathcal{R}_3, w, C) \vee \mathrm{Match}(\mathcal{R}_2\mathcal{R}_3, w, C)$
        case $\langle \mathcal{R}_1^*, \mathcal{R}_2 \rangle$ do
            return
              $\mathrm{Match}(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2, w, C \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \vee \mathrm{Match}(\mathcal{R}_2, w, C)$
```

Observe that usually in matching engines the match is successful even if just a prefix of the word matches the regex [21, 22], and this is known as *submatch* or *partial match* semantics. We can easily model this behaviour by appending $\Sigma^*$ at the end of regexes [9]. In the rest of the paper, for the sake of simplicity and without loss of generality, we assume that the match is successful only if the entire word is matched (*fullmatch* semantics). Since real-world matching engines use the partial match semantics, our implementation assumes instead by default such semantics. The translation between the two simply rewrites $\mathcal{R} \in \mathbb{R}$ as $\mathcal{R}\Sigma^*$.

# 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 whether 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 $\varepsilon$-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.

$$\begin{array}{lll} r(\varepsilon) \triangleq \varepsilon & r(\mathcal{R}_1|\mathcal{R}_2) \triangleq r(\mathcal{R}_1)|r(\mathcal{R}_2) & r(\mathcal{R}_1^*) \triangleq r(\mathcal{R}_1)^* \\ r(a) \triangleq a & r(\mathcal{R}_1\mathcal{R}_2) \triangleq r(\mathcal{R}_1)r(\mathcal{R}_2) & r(\mathcal{R}_1^{\overline{*}}) \triangleq r(\mathcal{R}_1)^* \end{array}$$

We then define the set of *actions* as $\mathbb{A} \triangleq \{\oplus, \oplus, \circledast, \bigcirc\} \cup \{ \text{🛆}_a \mid a \in \Sigma \}$. 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.

$$\begin{array}{cc} \langle a, aw\rangle \xrightarrow{\text{🛆}_a} \langle \varepsilon, w\rangle & \langle a\mathcal{R}_1, aw\rangle \xrightarrow{\text{🛆}_a} \langle r(\mathcal{R}_1), w\rangle \\ \langle \mathcal{R}_1|\mathcal{R}_2, w\rangle \xrightarrow{\oplus} \langle \mathcal{R}_1, w\rangle & \langle (\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, w\rangle \xrightarrow{\oplus} \langle \mathcal{R}_1\mathcal{R}_3, w\rangle \\ \langle \mathcal{R}_1|\mathcal{R}_2, w\rangle \xrightarrow{\oplus} \langle \mathcal{R}_2, w\rangle & \langle (\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3, w\rangle \xrightarrow{\oplus} \langle \mathcal{R}_2\mathcal{R}_3, w\rangle \\ \langle \mathcal{R}_1^*, w\rangle \xrightarrow{\circledast} \langle \mathcal{R}_1\mathcal{R}_1^{\overline{*}}, w\rangle & \langle \mathcal{R}_1^*\mathcal{R}_2, w\rangle \xrightarrow{\circledast} \langle \mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, w\rangle \\ \langle \mathcal{R}_1^*, w\rangle \xrightarrow{\bigcirc} \langle \varepsilon, w\rangle & \langle \mathcal{R}_1^*\mathcal{R}_2, w\rangle \xrightarrow{\bigcirc} \langle \mathcal{R}_2, w\rangle \end{array}$$

The transition relation describes all possible choices of the matching engine according to the state. Observe that with the $\circledast$ 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. Observe that the transition relation describes all possible actions that Algorithm 1 might perform in a particular state.

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 $\langle \mathcal{R}_0, w_0\rangle \in \mathbb{S}$.

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

$$\begin{aligned}
\{\, &\langle a^*, a\rangle, \langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\bigcirc} \langle \varepsilon, a\rangle \,\}
\end{aligned}$$

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

$$\begin{aligned}
\{\, &\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\bigcirc} \langle \varepsilon, a\rangle \,\}
\end{aligned}$$

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

$$\begin{aligned}
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\bigcirc} \langle \varepsilon, a\rangle
\end{aligned}$$

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

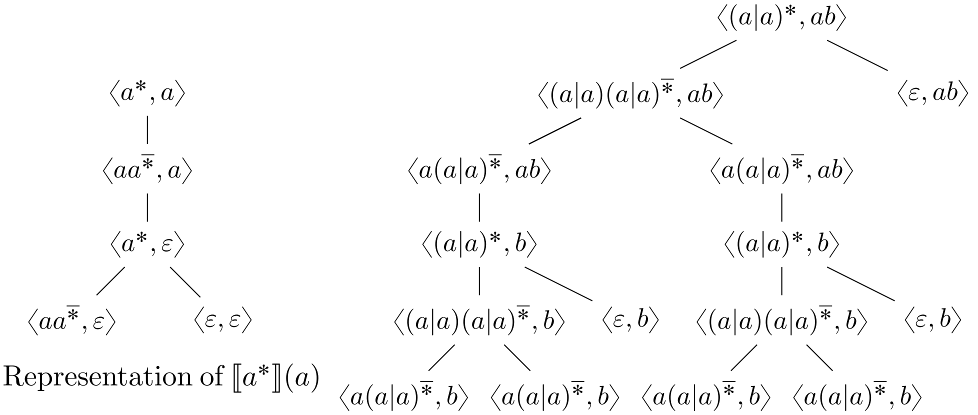
$$\begin{aligned}
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, \varepsilon\rangle,\\
&\langle a^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}, a\rangle \xrightarrow{\textcircled{\scriptsize$\triangle$}_a} \langle a^*, \varepsilon\rangle \xrightarrow{\bigcirc} \langle \varepsilon, \varepsilon\rangle
\end{aligned}$$

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

Figure 2: Intermediate steps to obtain the matching tree semantics

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 $\circledast$. This order assigns higher priority to the traces that choose to expand the left branch of the alternative or to expand the body of the star, which is the standard behaviour of matching engines. 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



$\langle a^*, a\rangle$
$\langle aa^{\overline{*}}, a\rangle$
$\langle a^*, \varepsilon\rangle$
$\langle aa^{\overline{*}}, \varepsilon\rangle$
$\langle \varepsilon, \varepsilon\rangle$
$\langle (a|a)^*, ab\rangle$
$\langle (a|a)(a|a)^{\overline{*}}, ab\rangle$
$\langle \varepsilon, ab\rangle$
$\langle a(a|a)^{\overline{*}}, ab\rangle$
$\langle a(a|a)^{\overline{*}}, ab\rangle$
$\langle (a|a)^*, b\rangle$
$\langle (a|a)^*, b\rangle$
$\langle (a|a)(a|a)^{\overline{*}}, b\rangle$
$\langle \varepsilon, b\rangle$
$\langle (a|a)(a|a)^{\overline{*}}, b\rangle$
$\langle \varepsilon, b\rangle$
$\langle a(a|a)^{\overline{*}}, b\rangle$
$\langle a(a|a)^{\overline{*}}, b\rangle$
$\langle a(a|a)^{\overline{*}}, b\rangle$
$\langle a(a|a)^{\overline{*}}, b\rangle$

(a) Representation of $[\![a^*]\!](a)$

(b) Representation of $[\![(a|a)^*]\!](ab)$

Figure 3: Examples of matching trees. Figure 3(a) represents a tree in which the matching is successful, while in Figure 3(b) the matching fails.

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 the 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 $S$.

**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 some examples of matching trees. 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 $\mathrm{lvs}(t)$. We define the *language recognized by* $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ as follows.

$$\mathcal{L}(\mathcal{R}) \triangleq \{\, w \in \Sigma^* \mid \langle \varepsilon, \varepsilon\rangle \in \mathrm{lvs}([\![\mathcal{R}]\!](w)) \,\}$$

We now give the definition of ReDoS vulnerability, using the one that firstly appeared in [10], 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 $M_{\mathcal{R}} \in \Omega(2^n)$.*

We now prove that the height of a matching tree $[\![\mathcal{R}]\!](w)$ is always $O(|w|)$. We define the *frontier* of a state $\langle \mathcal{R}, w \rangle$ as the (possibly empty) ordered sequence of states that are reached after matching the first character of the word $w$. More formally, the frontier $f : \mathbb{S} \to \mathbb{S}^*$ is the sequence of states $f(\langle \mathcal{R}, aw \rangle) \triangleq \langle r(\mathcal{R}_1), w \rangle, \ldots, \langle r(\mathcal{R}_n), w \rangle$, where $[\![a\mathcal{R}_1]\!](aw), \ldots, [\![a\mathcal{R}_n]\!](aw)$ 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, $f(\langle (a|a)^*, ab \rangle) = \langle (a|a)^*, b \rangle, \langle (a|a)^*, b \rangle$ (see Figure 3(b)). We define $f(\langle \mathcal{R}, \varepsilon \rangle)$ as the empty sequence. We abuse the notation and we generalize the frontier to sequences of states: $f(\langle \mathcal{R}_1, w \rangle, \ldots, \langle \mathcal{R}_n, w \rangle)$ is the ordered concatenation of the frontiers $f(\langle \mathcal{R}_1, w \rangle), \ldots, f(\langle \mathcal{R}_n, w \rangle)$. Furthermore, $f^0(\langle \mathcal{R}, w \rangle) \triangleq \langle \mathcal{R}, w \rangle$, and for $n \geqslant 1$, $f^n(\langle \mathcal{R}, w \rangle) \triangleq f \circ f^{n-1}(\langle \mathcal{R}, w \rangle)$. The following lemma formalizes the intuition that the length of input words is an upper bound for the height of matching trees.

**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, $h = \Theta(m)$. Let $n \leqslant m$, and observe that the words in the states of $f^n(\langle \mathcal{R}, w \rangle)$ have exactly $|w| - n$ characters. By definition of $f$, when $n = |w|$ the next frontier must be empty. This implies that $m$ cannot be greater than $|w|$, so that $h = O(|w|)$. □

# 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 49 false positives over 74,669 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

**Algorithm 2:** Compute $\mathcal{M}_2(\mathcal{R})$

```
function M2($\mathcal{R} : \mathbb{R}$) $\to \mathbb{R}^{\perp}$
    return M2-rec($\mathcal{R}, \varnothing$)
function M2-rec($\mathcal{R} : \mathbb{R}^{\mathcal{T}}, E : \wp(\mathbb{R}^{\mathcal{T}})$) $\to \mathbb{R}^{\perp}$
    if $\mathcal{R} \in E$ then
        return $\perp$
    switch $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle$ do
        case $\langle \varepsilon, \varepsilon \rangle \vee \langle \mathcal{R}_1^{\overline{*}}, \mathcal{R}_2 \rangle$ do
            return $\perp$
        case $\langle a, \mathcal{R}_1 \rangle$ do
            return $a \cdot$ M2-rec($r(\mathcal{R}_1), E$)
        case $\langle \mathcal{R}_1 | \mathcal{R}_2, \mathcal{R}_3 \rangle$ do
            $inter \leftarrow \mathcal{R}_1\mathcal{R}_3 \cap^r_{\not\varepsilon} \mathcal{R}_2\mathcal{R}_3$
            $l \leftarrow$ M2-rec($\mathcal{R}_1\mathcal{R}_3, E$)
            $r \leftarrow$ M2-rec($\mathcal{R}_2\mathcal{R}_3, E$)
            return $inter \cup^r l \cup^r r$
        case $\langle \mathcal{R}_1^*, \mathcal{R}_2 \rangle$ do
            $inter \leftarrow \mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\not\varepsilon} \mathcal{R}_2$
            $l \leftarrow$ M2-rec($\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}\}$)
            $r \leftarrow \mathcal{R}_1^* \cdot$ M2-rec($\mathcal{R}_2, E$)
            return $inter \cup^r l \cup^r r$
```

engine to explore all traces, effectively performing a ReDoS attack. This is the reason why $|[\![(a|a)^*]\!](a^n b)| = \Theta(2^n)$.

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 $\mathcal{L}(\mathcal{R}_1) = \mathcal{M}_2(\mathcal{R})$. 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 $\varepsilon$, and we denote it by $\mathcal{R}_1 \cap^r_{\not\varepsilon} \mathcal{R}_2$. It can be computed as $\not\varepsilon(\mathcal{R}_1) \cap^r \not\varepsilon(\mathcal{R}_2)$, where $\not\varepsilon : \mathbb{R}^{\mathcal{T}} \to \mathbb{R}^{\perp}$ removes $\varepsilon$ from the language of input regexes. The procedure is depicted in Algorithm 3.

The intuition behind M2 is that a word is matched in two different traces if

**Algorithm 3:** Remove $\varepsilon$ from $\mathcal{L}(\mathcal{R})$

```
function $\not\varepsilon(\mathcal{R} : \mathbb{R}^{\mathcal{T}}) \to \mathbb{R}^{\perp}$
    switch $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle$ do
        case $\langle \varepsilon, \varepsilon \rangle \vee \langle \mathcal{R}_1^{\overline{*}}, \mathcal{R}_2 \rangle$ do
            return $\perp$
        case $\langle a, \mathcal{R}_1 \rangle$ do
            return $a \cdot (r(\mathcal{R}_1))$
        case $\langle \mathcal{R}_1 | \mathcal{R}_2, \mathcal{R}_3 \rangle$ do
            return $\not\varepsilon(\mathcal{R}_1\mathcal{R}_3) \cup^r \not\varepsilon(\mathcal{R}_2\mathcal{R}_3)$
        case $\langle \mathcal{R}_1^{*}, \mathcal{R}_2 \rangle$ do
            return $\not\varepsilon(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup^r \not\varepsilon(\mathcal{R}_2)$
```

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, is encountered for the second time, the function returns $\perp$. 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 $\varepsilon$-matching loops; the second guarantees termination of the M2-rec function.

**Theorem 1** (Correctness of M2)**.** *Let* $\mathcal{R} \in \mathbb{R}$*.*

$$\mathcal{L}(\mathrm{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$$

In A we give the detailed proof of Theorem 1, while here we show the proof sketch and main intuition.

*Proof (Sketch).* We show that in both the base and the recursive case the function M2-rec computes $\mathcal{M}_2$. Since M2 immediately calls M2-rec, this is equivalent to proving $\mathcal{L}(\mathrm{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$. In the base case of M2-rec, there are no recursive calls, and there are only three possible cases. If $\mathcal{R} \in E$, then we already analyzed

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

$\mathcal{R}$, and we simply return $\bot$. If $\mathcal{R} = \varepsilon$, since the words in $\mathcal{M}_2$ are non-empty, we correctly return $\bot$. If $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle \mathcal{R}_1^{\overline{*}}, \mathcal{R}_2 \rangle$, there exist no word that can be matched in $\mathcal{R}$, so that we return $\bot$.

In the inductive case we have three other subcases that depend on $\mathcal{R}$.

- If $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle a, \mathcal{R}_1 \rangle$, we observe that the first constructor leads to only one possible action, that is matching the symbol $a$. Therefore, the words that can be matched in two different traces in $\mathcal{R}$ are those that start with the symbol $a$, and that can be matched in at least two different traces in $\mathcal{R}_1$, so that we return $a \cdot$ M2-rec$(r(\mathcal{R}_1), E)$. By inductive hypothesis, this is exactly $a \cdot (\mathcal{M}_2(r(\mathcal{R}_1)))$.

- If $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle \mathcal{R}_1 | \mathcal{R}_2, \mathcal{R}_3 \rangle$, the words that can be matched in two different traces are: (1) those that can be matched by both branches of the alternative and then can be matched in $\mathcal{R}_3$, namely $\mathcal{R}_1\mathcal{R}_3 \cap^r \mathcal{R}_2\mathcal{R}_3$; (2) those that can be matched in two different traces in the left branch, namely $\mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3)$; (3) those that can be matched in two different traces in the right branch, namely $\mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3)$. We can conclude by inductive hypothesis by observing that we return $(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_3, E) \cup^r \text{M2-rec}(\mathcal{R}_2\mathcal{R}_3, E)$.

- If $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle \mathcal{R}_1^*, \mathcal{R}_2 \rangle$, the words that can be matched in two different traces are: (1) those that can be matched by both expanding and not expanding the star, namely $\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2$; (2) those that can be matched in two different traces by expanding the body of the star, namely $\mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$; (3) those that can be matched in two different traces in $\mathcal{R}_2$ after matching a prefix in $\mathcal{R}_1^*$, namely $\mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)$. We can conclude by inductive hypothesis by observing that we return $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}\}) \cup^r \mathcal{R}_1^* \cdot \text{M2-rec}(\mathcal{R}_2, E)$.

□

**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* $\mathcal{M}_2((a|a)^*)$*.*

**Example 2.** *Heuristics-based tools often classify as dangerous regexes that have nested stars. In this example, we show how this pattern implies that the language of words that can be matched in at least two traces is non-empty. Consider the regex $(a^*)^*$ and the word $aa$. After matching the first character $a$, the matching engine reaches the state $\langle a^*(a^*)^*, a\rangle$, as shown by the following partial trace:*

$$\langle (a^*)^*, aa\rangle \xrightarrow{\circledast} \langle a^*(a^*)^{\overline{*}}, aa\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}(a^*)^{\overline{*}}, aa\rangle \xrightarrow{\textcircled{$\vartriangle$}_a} \langle a^*(a^*)^*, a\rangle$$

*In this configuration, it is possible to match the subsequent character $a$ by expanding either the left or the right star:*

$$\langle a^*(a^*)^*, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}(a^*)^*, a\rangle \xrightarrow{\textcircled{$\vartriangle$}_a} \langle a^*(a^*)^*, \varepsilon\rangle$$

$$\langle a^*(a^*)^*, a\rangle \xrightarrow{\bigcirc} \langle (a^*)^*, a\rangle \xrightarrow{\circledast} \langle a^*(a^*)^{\overline{*}}, a\rangle \xrightarrow{\circledast} \langle aa^{\overline{*}}(a^*)^{\overline{*}}, a\rangle \xrightarrow{\textcircled{$\vartriangle$}_a} \langle a^*(a^*)^*, \varepsilon\rangle$$

*This implies that the language of words that can be matched in at least two traces is non-empty. In general, nested stars can lead to this type of configuration in which words can be matched in two different concatenated stars. This implies that the regex might be dangerous, justifying the decision of heuristics-based tools to classify regexes with nested stars as vulnerable.*

*When analyzing $(a^*)^*$, after three recursive calls,* M2-rec *reaches the regex $a^*(a^*)^*$ and returns $a^* \cap^{r}_{\not\varepsilon} (a^*)^* =_{\mathcal{L}} a^+$. This regex is then concatenated to the prefix that makes it possible to reach the configuration $a^*(a^*)^*$, namely $a$. Overall, the language of words that can be matched in at least two different traces is $a{\cdot}a^+$.*

Intuitively, if there is no word that is matched in two different traces, there is no ambiguity, and the matching is linear in the length of the input words in the worst case. In Lemma 2, we prove this intuition.

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

$$\mathcal{M}_2(\mathcal{R}) = \varnothing \implies |[\![\mathcal{R}]\!](w)| = O(|w|)$$

*Proof.* Let $t'$ be the subtree from the root $\langle \mathcal{R}, w\rangle$ to the nodes in the frontier $f(\langle \mathcal{R}, w\rangle)$. Observe that the nodes in $f(\langle \mathcal{R}, w\rangle)$ are the only ones that possibly have subtrees outside the portion that we are considering: all the others are either internal nodes in $t'$ or do not have children. Observe also that the number of nodes in $t'$ does not depend on $|w|$, but just on $\mathcal{R}$ and the first character of $w$, if there is any.

We define the set of *reachable regexes* $\text{rch} : \mathbb{R}^{\mathcal{T}} \to \wp(\mathbb{R}^{\mathcal{T}})$ as follows.

$$\text{rch}(\mathcal{R}) \triangleq \{\, \mathcal{R}' \in \mathbb{R}^{\mathcal{T}} \mid \exists w_1, w_2 \in \Sigma^*, \exists t \in \mathcal{T}(\langle \mathcal{R}, w_1 w_2\rangle) : \ell(t) = \langle \mathcal{R}', w_2\rangle \,\}$$

The 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 $i \in \{1, \ldots, |w|\}$:

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

Since the width of the matching tree grows as the size of the frontiers, this implies that the width of the matching tree is $O(|\operatorname{rch}(\mathcal{R})|)$. By Lemma 1, the height of the matching tree is at most linear in the length of the word, so that $|[\![\mathcal{R}]\!](w)| = O(|w| \cdot |\operatorname{rch}(\mathcal{R})|)$. Since $|\operatorname{rch}(\mathcal{R})|$ does not depend on $|w|$, we conclude that $|[\![\mathcal{R}]\!](w)| = O(|w|)$. □

To understand how we take advantage of M2, consider a regex $\mathcal{R}^*$ such that $\mathrm{M2}(\mathcal{R}^*) \neq_{\mathcal{L}} \varnothing$. In this case, the set of words that are matched with at least two traces in $\mathcal{R}^*$ is not empty. Let $w \in \mathcal{L}(\mathrm{M2}(\mathcal{R}^*))$. 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 $n \geqslant 1$, $w^n \in \mathcal{L}(\mathrm{M2}(\mathcal{R}^*))$. This implies that the words in $\mathrm{M2}(\mathcal{R}^*)$ 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 $\mathrm{M2}(\mathcal{R}^*)\cdot\overline{\mathcal{R}^*\mathcal{S}}^r$ 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 $\mathcal{E} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{\perp}$ 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 $\mathcal{R}$.

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

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 $\mathcal{P}\cdot\mathcal{R}_1^*\cdot \mathrm{M2}(\mathcal{R}_1^*)\cdot\overline{\mathcal{R}_1^*\mathcal{S}}^r$. As discussed previously,

$\mathrm{M2}(\mathcal{R}_1^*)\overline{\mathcal{R}_1^*\mathcal{S}}^r$ 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 3.** *Consider* $\mathcal{E}((a|a)^*, \varepsilon, \varepsilon)$*.*

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

*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 a's 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. More precisely, the matching is at most polynomial in the number of stars that syntactically appear in the regex.

**Theorem 2** (Soundness)**.** *Let* $\mathcal{R} \in \mathbb{R}$*.*

$$\mathcal{E}(\mathcal{R}, \varepsilon, \varepsilon) =_{\mathcal{L}} \bot \implies |[\![\mathcal{R}]\!](w)| = O(|w|^{s(\mathcal{R})})$$

*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 $|[\![\mathcal{R}]\!](w)| = O(1)$. This can be seen as a consequence of the fact that $\mathcal{L}(\mathcal{R})$ is finite.

The inductive case is $s(\mathcal{R}) \geqslant 1$. The only case that we consider is when $\mathrm{hd}(\mathcal{R}) = \mathcal{R}_1^*$ and $\mathrm{tl}(\mathcal{R}) = \mathcal{R}_2$. All other cases can be reduced to this: regex constructors that are not stars can match only a constant number of characters before reaching a star. Observe that by definition of $\mathcal{E}$, $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \varepsilon, \varepsilon) =_{\mathcal{L}} \bot$ implies $\mathcal{E}(\mathcal{R}_2, \mathcal{R}_1^*, \varepsilon) =_{\mathcal{L}} \bot$. 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 obtain the following.

$$|[\![\mathcal{R}_2]\!](w)| = O(|w|^{s(\mathcal{R}_2)}) = O(|w|^{s(\mathcal{R}_1^*\mathcal{R}_2)-1})$$

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 polynomial in $|w'|$, which implies that the size is at most polynomial 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 $\mathrm{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \perp$ 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. We already observed that all those subtress have size $O(|w|^{s(\mathcal{R}_1^*\mathcal{R}_2)-1})$. Therefore, we obtain:

$$\begin{aligned}|[\![\mathcal{R}_1^*\mathcal{R}_2]\!](w)| &= O(|w|) + \sum_{i=1}^{|w|} O(|w|^{s(\mathcal{R}_1^*\mathcal{R}_2)-1}) \\ &= O(|w|) + |w|O(|w|^{s(\mathcal{R}_1^*\mathcal{R}_2)-1}) \\ &= O(|w|^{s(\mathcal{R}_1^*\mathcal{R}_2)})\end{aligned}$$

This proves the theorem. Observe that actually $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \varepsilon, \varepsilon) =_{\mathcal{L}} \perp$ can be caused not exclusively by $\mathrm{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \perp$, but also by $\overline{\mathcal{R}_1^*\mathcal{R}_2}^r =_{\mathcal{L}} \perp$. The only language that has as complement the empty language is $\Sigma^*$, 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. □

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 74,669 regexes taken from real-world use cases, this happens only in 49 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.

As discussed in Section 2.5, in this paper we assume that the match is successful only if the entire word matches the regular expression (fullmatch semantics). Nevertheless, matching engines usually consider the match to be valid even if just a prefix of the word matches the regex (partial match semantics). To simulate this behaviour, we can simply append $\Sigma^*$ at the end of the patterns. Observe that the complement of the universal language is $\perp$, so that if $\Sigma^*$ is the only suffix of a dangerous star, the exponential behaviour cannot be triggered. As discussed in this section, this is because there exists no suffix that can make the match fail. The implication is that patterns that are dangerous in the fullmatch semantics, can be harmless in the partial match semantics. Since the latter is the one used in matching engines, our tool `rat` assumes it by default, but the translation between the two is trivial.

# 5 Experimental Evaluation

To assess the usefulness of the analysis we put forward, we implemented it in the `rat` [17] tool (**R**eDoS **A**bstract **T**ester) in less than 5000 lines of OCaml code, and we compared it to seven 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 [12], composed of: (1) 2,992 patterns from the Regexlib platform [41]; (2) 12,499 patterns from the Snort platform [45]; (3) 13,597 patterns extracted from 3,898 Python projects on Github in [2]. To them, we added 63,352 regexes extracted from modules in the `pypi` package manager [46] by Davis et al. [47]. From the dataset, we removed the regexes that were not properly sanitized (e.g., that contained non-printable characters) and we removed duplicates, obtaining 74,669 regexes. To

| | **Type** | **Sound** | **Complete** | **Language** | **Deterministic** |
|---|---|---|---|---|---|
| `rat` | *static, semantic* | ✓ | ✗ | ✓ | ✓ |
| `ReScue` [12] | *dynamic* | ✗ | ✓ | ✗ | ✗ |
| `rexploiter` [11] | *static, semantic* | ✗ | ✗ | ✓ | ✓ |
| `rsa` [10] | *static, semantic* | ✓ | ✗ | ✗ | ✓ |
| `rsa-full` [10] | *static, semantic* | ✓ | ✓ | ✗ | ✓ |
| `rxxr2` [9] | *static, semantic* | ✗ | ✗ | ✗ | ✓ |
| `safe-regex` [14] | *static, heuristic* | ✗ | ✗ | ✗ | ✓ |
| `regexploit` [15] | *static, heuristic* | ✗ | ✗ | ✗ | ✓ |
| `redos-detector` [16] | *static, heuristic* | ✗ | ✗ | ✗ | ✓ |

Table 2: Attributes of the detectors

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` [12], `rexploiter` [11], `rsa` [10], `rxxr2` [9], `safe-regex` [14], `regexploit` [15] and `redos-detector` [16]. 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. The detectors `safe-regex`, `regexploit` and `redos-detector` are heuristics-based, and they do not offer any guarantees about the soundness or the completeness of the analysis. In Section 6, we discuss the details of each approach, and in Table 2 we summarize the characteristics of the tools. While attributes reported in Table 2 summarize the expected behaviour, we found that in practice some detectors do not match the underlying theoretical results. For instance, while `rsa-full` should be sound and complete, we found that it reports both false positives and false negatives. If a detector can extract the language of dangerous words (opposed to a single exploit string) we mark the **Language** column with ✓. Static detectors are divided into semantics-based and heuristics-based tools.

## 5.1 Precision Comparison.

We take advantage of the evaluation technique used in [12], which, to the best of our knowledge, is the only article that compares the precision of different ReDoS detectors. We analyze each regex with the detectors setting an individual timeout

| | **OK** | **FP** | **FN** | **OOT** | **RTE** | **SKIP** | **TIME** |
|---|---:|---:|---:|---:|---:|---:|---:|
| `rat` | 67,052 | 49 | 0 | 178(21) | 0(0) | 7,390(13) | 1:57:20 |
| `rxxr2` | 60,794 | 93 | 7 | 10(2) | 0(0) | 13,765(23) | 0:09:29 |
| `ReScue` | 33,531 | 0 | 40 | 32,208(43) | 0(0) | 8,890(34) | 325:00:26 |
| `rsa` | 57,269 | 193 | 1 | 789(47) | 240(35) | 16,177(42) | 18:48:02 |
| `rsa-full` | 54,857 | 134 | 1 | 3,138(55) | 400(43) | 16,139(42) | 38:11:07 |
| `rexploiter` | 53,931 | 28 | 180 | 328(1) | 0(0) | 20,202(104) | 9:12:34 |
| `safe-regex` | 61,272 | 13,376 | 21 | 0(0) | 0(0) | 0(0) | 0:15:40 |
| `regexploit` | 74,050 | 56 | 140 | 2(0) | 0(0) | 421(14) | 0:03:41 |
| `redos-detector` | 45,694 | 14,218 | 6 | 2(1) | 0(0) | 14,749(92) | 0:52:27 |

Table 3: Evaluation results

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 eight 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. Nevertheless, since we include `ReScue` in the comparison, which is a nondeterministic detector, these numbers might vary slightly in different runs. We classified as vulnerable 316 regexes.

Heuristics-based detectors do not have semantics information about the attack language and they do not perform dynamic detection either, so that they can rarely report useful exploit strings. We experienced some difficulties in extracting exploit strings from those analyzers. When possible, we extracted attack strings based on a best-effort implementation.

In Table 3, we report the results of the comparison. The columns correspond to: number of correctly classified regexes (**OK**); false positives (**FP**); false negatives (**FN**); out of time (**OOT**); runtime errors (**RTE**); skipped (**SKIP**) (i.e., not parsed); total runtime displayed as H:MM:SS (**TIME**). For out of time events, runtime errors, and skipped regexes, we report in parentheses how many regexes in the total number are vulnerable.

Compared to other static analyzers, `rat` reports a relatively low number of false positives: 49 over the 67,074 regexes that it parses. The only static analyzer that reports fewer false positives than `rat` is `rexploiter`, that on the other hand reports respectively 180 false negatives. Furthermore, `rexploiter` skips 20,202 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.

If we do not consider heuristics-based tool, `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. Heuristics-based detectors can analyzer a higher number of regexes: `regexploit` and `safe-regex` skip respectively only 421 and 0 regular expressions. Since these tools do not offer guarantees about the soundness or the completeness of the analysis, they can analyze a wide variety of constructs by simply ignoring them. On the other hand, we observe that `safe-regex` parses and analyzes regular expressions that, to the best of our knowledge, are not accepted by any matching engine, for instance `(a**`. The high number of false positives reported by `safe-regex` and `redos-detector` makes it difficult to use them in practice. In fact, they raise respectively 13,376 and 14,218 false alarms.

## 5.2 Performance Comparison.

In case a detector runs out of time for a few regexes, the total runtime in Table 3 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 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 $74,669-x$ 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 66,926 regexes over the 67,074 that it parses in less than one second each ($\sim$ 99.78%). 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 $x$. Even though the total runtime to analyze the whole dataset for `redos-detector` is lower than `rat`'s, the plot shows how our tool performs significantly better on average. The same holds for `safe-regex`: in 82,8% of the cases `rat` is faster. The `regexploit` tool performs better than our analyzer, at the cost of raising 140 false negatives, meaning that it does not detect more than one third of the vulnerabilities. While `rxxr2` is generally faster than `rat`, we remark that `rat` is performing a strictly more expressive analysis by returning the

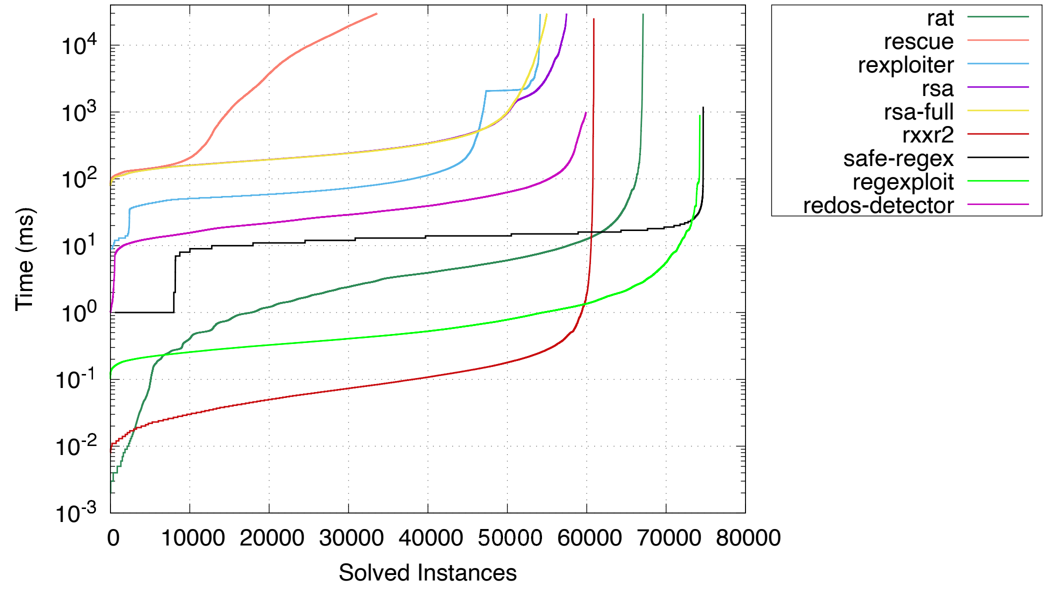


Figure 4: Survival plot with a logarithmic $y$ axis and linear $x$ axis

language of dangerous words. Furthermore, according to Table 3, `rxxr2` is not performing a sound analysis either. We also remark that `rat` analyzes 6,375 more regexes than `rxxr2`.

## 5.3 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` three major performance improvements.

- *Character Classes Representation.* Character classes are features commonly used by programmers. For example, `\d` is a shortcut for $0|1|\ldots|9$. 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, $0|\ldots|9$ has 19 constructs, while $\{\,0,\ldots,9\,\}$ 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.

- *Emptiness Check.* The last step in the analysis is to check if the computed attack language is empty. We decided to take advantage of the algorithm based on *derivatives* put forward in [48], which, as the results of our experiments confirm, efficiently performs the emptiness check. The framework described in [48] uses *extended regular expressions* with symbolic intersection and symbolic complement, so that it can be effortlessly integrated into our implementation.

We conducted an analysis to determine the number of regular expressions that produce false positives in both `rat` and other tools. Our investigation found that `rxxr2` and `rsa` share respectively 21 and 26 false positives with `rat`. This overlap is significant and can be attributed to similarities in the approaches used by these detectors. Typically, automata-based tools leverage analysis techniques to detect nondeterministic transitions in the loops of the automata. Our algorithm M2 performs a similar analysis on the stars of regular expressions, as it detects the language of words that can be matched in two different traces. As stars in regular expressions are often transformed into loops in automata, we can account for the shared false positives between `rat` and automata-based tools.

Upon examining the false positives reported by other tools, we found no correlation with `rat`. In the case of `safe-regex` and `redos-detector`, the number of false alarms generated was too high to draw meaningful conclusions. In the case of `regexploit`, the overlap is limited to 10 false positives, while `ReScue` cannot report false alarms. Although `rexploiter` employs an automata-based algorithm, only two false positives were shared between the tool and `rat`. This finding highlights that the translation algorithm used by `rexploiter` fails to preserve the structure, and therefore the vulnerabilities, of regular expressions.

In our experimental evaluation, we did not find any recurring pattern in the false positives raised by `rat`. By considering a large set of regular expressions that result in false positives, we might build a database of rules to improve the precision of the analysis in specific cases. Nonetheless, the soundness guarantee offered by our theoretical framework does not trivially hold if we add human-crafted ad-hoc rules to our analyzer. As a result, any rule that is used must be proved to preserve the soundness of the analysis.

# 6 Related Work

In this section we discuss related work. In particular, we describe existing ReDoS detection techniques, ReDoS mitigation frameworks, and the link between our semantics and regular expression derivatives.

## 6.1 Semantics-Based Static ReDoS Detection

Wüstholz et al. [11] 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 [49] to the vulnerable regex $(a^*)^*$ we obtain a non-vulnerable automaton (with respect to [11, 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 [9]. It is the successor of `rxxr` [8], 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) [28, 50] has been leveraged by Weideman et al. [10] 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 [51] to determine the *degree of ambiguity* [52], 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 [10].

## 6.2 Dynamic ReDoS Detection

A radically different approach to ReDoS detection is dynamic analysis. The `ReScue` tool [12] 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.

## 6.3 Heuristics-Based Static ReDoS Detection

Heuristics-based static analyzers try to report vulnerabilities by matching potentially dangerous patterns against the constructors of a regular expression. For instance, `safe-regex` [14] checks that regexes do not present nested stars. It is easy to craft an example for which this rule raises a false positive: the regex `(a*)*.*` has two nested stars, but since there is no suffix that can make the matching fail (`.*` accepts the universal language), the regex is not dangerous. Nevertheless, `safe-regex` reports that the regex is dangerous, effectively raising a false positive. In our experiments, we also found that both `safe-regex` and `regexploit` raise a false negative when analyzing the regex `<project(.|\s)*?>`, as they do not detect the exponential vulnerability. The exponential behaviour can be triggered by using as exploit string `<project` followed by a sequence of space characters, since spaces can be matched by both branches of the alternative `(.|\s)`. Heuristics-based analyzers do not have semantic information about the attack language, and they do not perform dynamic testing either. In our experiments, we observed that these tools report a high number of false positives and false negatives. The heuristics employed by `safe-regex` [14], `regexploit` [15] and `redos-detector` [16] are not formalized in any work, and they can potentially change in the future. Not having

semantic information also implies that it is impossible for this class of detectors to differentiate the type of the vulnerabilities reported, namely if the matching is exponential or superlinear.

## 6.4 ReDoS Mitigation

Recently, many techniques have been proposed to mitigate ReDoS attacks. Cody-Kenny et al. [53] use genetic programming to substitute vulnerable regexes with safe ones. Li et al. [54] and Pan et al. [55] put forward techniques for automatic regex repair based on examples. In [56] 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 [57, 58] and state-merging algorithms [59] have also been proposed to speedup the matching.

## 6.5 Regular Expression Derivatives

Derivatives-based matching [38, 39, 40] is a technique to perform regular expression matching. It relies on the fundamental concept of *derivative of a regular expression*. In general, given a symbol $a$, the derivative of a regex $\mathcal{R}$ with respect to $a$ is a regex that recognizes only those suffixes of strings with a leading $a$ accepted by $\mathcal{R}$. Brzozowski's derivatives [38] are related to DFAs, while Antimirov's partial derivatives [39] are related to NFAs, and both can be leveraged to perform regex matching. Matching engines in widely used programming languages do not use derivatives-based matching, as they rely on backtracking algorithms [21, 22, 28].

There are some similarities between our tree semantics and Brzozowski's derivatives. The connection lies in the fact that when we match the first character from the state $\langle \mathcal{R}, aw \rangle$, the regexes in the frontier (see page 13) of $\langle \mathcal{R}, aw \rangle$ recognize the same language accepted by the derivative of $\mathcal{R}$ with respect to the character $a$.

Nevertheless, there are substantial differences between the two approaches. In fact, our semantics is designed to capture the exact states explored by the matching engine, and in which order they appear. For instance, we can observe that after matching the first $a$ starting from $\langle (a|a)^*, ab \rangle$, we explore the state $\langle (a|a)^*, b \rangle$ exactly twice. This would not be possible by using derivatives, as they do not enjoy a notion of order over the expanded regexes. Furthermore, to mimic the behaviour of matching engines we added the *closed star* constructor, which is not needed in derivatives. Since regex derivatives cannot precisely capture the state of the matcher, they are not suitable to formally describe and reason about ReDoS vulnerabilities.

# 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 seven 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 number of false positives, `rat` is the only ReDoS detector that did not report false negatives.

In future work, 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. We would also like to use the matching semantics to design a detector for superlinear ReDoS vulnerabilities. Similarly to the exponential case, we expect that an approach based on regular expressions can lead to an efficient and sound analysis also for superlinear vulnerabilities. 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.

*Acknowledgement:* This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 - MOPSA.

# References

[1] S. A. Crosby, D. S. Wallach, Denial of service via algorithmic complexity attacks, in: USENIX Security Symposium, USENIX Association, 2003. `doi:10.1007/11506881\_10`.

[2] C. Chapman, K. T. Stolee, Exploring regular expression usage and context in Python, in: International Symposium on Software Testing and Analysis, ISSTA, ACM, 2016, pp. 282–293. `doi:10.1145/2931037.2931073`.

[3] C. Staicu, M. Pradel, Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers, in: USENIX Security Symposium, USENIX Association, 2018, pp. 361–376.

[4] L. G. M. IV, J. Donohue, J. C. Davis, D. Lee, F. Servant, Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions, in: International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 415–426. `doi:10.1109/ASE.2019.00047`.

[5] Stack overflow outage postmortem, accessed: 2023-03-08 (2016).
URL https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016

[6] Cloudflare's outage postmortem, accessed: 2023-03-08 (2019).
URL https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/

[7] National vulnerability database: CVE-2020-3899, accessed: 2023-03-08 (2020).
URL https://nvd.nist.gov/vuln/detail/CVE-2020-3899

[8] J. Kirrage, A. Rathnayake, H. Thielecke, Static analysis for regular expression denial-of-service attacks, in: International Conference of Network and System Security, NSS, Vol. 7873 of Lecture Notes in Computer Science, Springer, 2013, pp. 135–148. `doi:10.1007/978-3-642-38631-2\_11`.

[9] A. Rathnayake, H. Thielecke, Static analysis for regular expression exponential runtime via substructural logics, CoRR abs/1405.7058 (2014). `arXiv:1405.7058`.

[10] N. Weideman, B. van der Merwe, M. Berglund, B. W. Watson, Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA, in: International Conference on Implementation and Application of Automata, CIAA, Vol. 9705 of Lecture Notes in Computer Science, Springer, 2016, pp. 322–334. `doi:10.1007/978-3-319-40946-7\_27`.

[11] V. Wüstholz, O. Olivo, M. J. H. Heule, I. Dillig, Static detection of dos vulnerabilities in programs that use regular expressions, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Vol. 10206 of Lecture Notes in Computer Science, 2017, pp. 3–20. `doi:10.1007/978-3-662-54580-5\_1`.

[12] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, J. Lu, ReScue: crafting regular expression DoS attacks, in: International Conference on Automated Software Engineering, ASE, ACM, 2018, pp. 225–235. `doi:10.1145/3238147.3238159`.

[13] The SonarSource tool, accessed: 2023-03-08.
URL https://www.sonarsource.com/

[14] The safe-regex tool, accessed: 2023-03-08.
URL https://github.com/substack/safe-regex

[15] The regexploit tool, accessed: 2023-03-08.
URL https://github.com/doyensec/regexploit

[16] The redos-detector tool, accessed: 2023-03-08.
URL https://github.com/tjenkinson/redos-detector

[17] F. Parolini, A. Miné, rat - ReDoS Abstract Tester (2022).
URL https://github.com/parof/rat

[18] F. Parolini, A. Miné, Sound static analysis of regular expressions for vulnerabilities to denial of service attacks, in: Y. Aït-Ameur, F. Crăciun (Eds.), 16th International Symposium on Theoretical Aspects of Software Engineering (TASE 2022), Springer International Publishing, 2022, pp. 73–91. doi:10.1007/978-3-031-10363-6_6.

[19] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation, 3rd Edition, Pearson international edition, Addison-Wesley, 2007.

[20] M. Lam, R. Sethi, J. Ullman, A. Aho, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[21] J. E. F. Friedl, Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.), O'Reilly, 2006.
URL https://www.oreilly.com/library/view/mastering-regular-expressions/0596528124/

[22] F. López, V. Romero, Mastering Python Regular Expressions, Packt Publishing Ltd, 2014.
URL https://www.packtpub.com/product/mastering-python-regular-expressions/9781783283156

[23] Rust's regex matching engine, accessed: 2023-03-08.
URL https://github.com/rust-lang/regex

[24] Rust's regex module documentation, accessed: 2023-03-08.
URL https://docs.rs/regex/latest/regex/

[25] Google's RE2 matching engine, accessed: 2023-03-08.
URL https://github.com/google/re2

[26] V8's regex matching engine, accessed: 2023-03-08.
URL https://github.com/v8/v8/tree/11.3.116/src/regexp

[27] V8 new matching engine announcement, accessed: 2023-03-08.
URL https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html

[28] M. Berglund, F. Drewes, B. van der Merwe, Analyzing catastrophic backtracking behavior in practical regular expression matching, in: Automata and Formal Languages, AFL, Vol. 151 of EPTCS, 2014, pp. 109–123. `doi:10.4204/EPTCS.151.7`.

[29] Java's regex matching engine, accessed: 2023-03-08.
URL https://github.com/openjdk/jdk/tree/jdk8-b120/jdk/src/share/classes/java/util/regex

[30] Php's regex matching engine, accessed: 2023-03-08.
URL https://github.com/php/php-src/tree/php-8.2.3/ext/pcre

[31] PCRE2 regex engine documentation, accessed: 2023-03-08.
URL https://www.pcre.org/current/doc/html/pcre2pattern.html

[32] Perl's regex matching engine, accessed: 2023-03-08.
URL https://github.com/Perl/perl5/blob/v5.37.9/regexec.c

[33] Perl's regex module documentation, accessed: 2023-03-08.
URL https://perldoc.perl.org/perlre

[34] Python's regex matching engine, accessed: 2023-03-08.
URL https://github.com/python/cpython/tree/3.11/Lib/re

[35] Python's regex module documentation, accessed: 2023-03-08.
URL https://docs.python.org/3/library/re.html

[36] Ruby's regex matching engine, accessed: 2023-03-08.
URL https://github.com/ruby/ruby/blob/v3_2_1/re.c

[37] Ruby's regex module documentation, accessed: 2023-03-08.
URL https://ruby-doc.org/core-2.7.0/Regexp.html

[38] J. A. Brzozowski, Derivatives of regular expressions, Journal of the ACM 11 (4) (1964) 481–494. `doi:10.1145/321239.321249`.

[39] V. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theoretical Computer Science 155 (2) (1996) 291–319. `doi:https://doi.org/10.1016/0304-3975(95)00182-4`.

[40] S. Owens, J. Reppy, A. Turon, Regular-expression derivatives re-examined, Journal of Functional Programming 19 (2) (2009) 173–190. doi:10.1017/s0956796808007090.

[41] Regexlib database, accessed: 2023-03-08.
URL https://regexlib.com/

[42] A. Rathnayake, Semantics, analysis and security of backtracking regular expression matchers, Ph.D. thesis, University of Birmingham, UK (2015).
URL http://etheses.bham.ac.uk/6011/

[43] T. Petsios, J. Zhao, A. D. Keromytis, S. Jana, Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities, in: Conference on Computer and Communications Security, CCS, ACM, 2017, pp. 2155–2168. doi:10.1145/3133956.3134073.

[44] Node package manager, accessed: 2023-03-08.
URL https://www.npmjs.com/

[45] The snort database, http://www.snort.org/, accessed: 2023-03-08 (2020).

[46] The pypi packet manager, https://pypi.org/, accessed: 2023-03-08.

[47] J. C. Davis, C. A. Coghlan, F. Servant, D. Lee, The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale, in: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, ACM, 2018, pp. 246–256. doi:10.1145/3236024.3236027.

[48] J. Midtgaard, F. Nielson, H. R. Nielson, A parametric abstract domain for lattice-valued regular expressions, in: International Static Analysis Symposium, SAS, Vol. 9837 of Lecture Notes in Computer Science, Springer, 2016, pp. 338–360. doi:10.1007/978-3-662-53413-7_17.

[49] V. M. Glushkov, The abstract theory of automata, Russian Mathematical Surveys 16 (5) (1961) 1.

[50] M. Berglund, B. van der Merwe, On the semantics of regular expression parsing in the wild, Theoretical Computer Science 679 (2017) 69–82. doi:10.1016/j.tcs.2016.09.006.

[51] C. Allauzen, M. Mohri, A. Rastogi, 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) (2011) 883–904. doi:10.1142/s0129054111008477.

[52] A. Weber, H. Seidl, On the degree of ambiguity of finite automata, Theorertical Computer Science 88 (2) (1991) 325–349. doi:10.1016/0304-3975(91)90381-B.

[53] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, M. O'Neill, A search for improved performance in regular expressions, in: Genetic and Evolutionary Computation Conference, GECCO, 2017, pp. 1280–1287. doi:10.1145/3071178.3071196.

[54] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S. Cheung, H. Zhao, Flashregex: Deducing anti-redos regexes from examples, in: International Conference on Automated Software Engineering, ASE 2020, 2020, pp. 659–671. doi:10.1145/3324884.3416556.

[55] R. Pan, Q. Hu, G. Xu, L. D'Antoni, Automatic repair of regular expressions, Proceedings of the ACM on Programming Languages 3 (OOPSLA) (2019) 139:1–139:29. doi:10.1145/3360565.

[56] J. C. Davis, F. Servant, D. Lee, Using selective memoization to defeat regular expression denial of service (ReDoS), in: IEEE Symposium on Security and Privacy, SP, IEEE Computer Society, 2021, pp. 543–559. doi:10.1109/SP40001.2021.00032.

[57] C. Lin, C. Liu, S. Chang, Accelerating regular expression matching using hierarchical parallel machines on GPU, in: Global Communications Conference, GLOBECOM, 2011, pp. 1–5. doi:10.1109/GLOCOM.2011.6133663.

[58] X. Yu, M. Becchi, GPU acceleration of regular expression matching for large datasets: exploring the implementation space, in: Computing Frontiers Conference, CF, 2013, pp. 18:1–18:10. doi:10.1145/2482767.2482791.

[59] M. Becchi, S. Cadambi, Memory-efficient regular expression search using state merging, in: Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, 2007, pp. 1064–1072. doi:10.1109/INFCOM.2007.128.

# A Proof of Correctness of M2 (Theorem 1)

In this section we prove the correctness of Algorithm 2, namely we show $\mathcal{L}(\text{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$ (see Theorem 1). Since M2 immediately calls M2-rec, we actually prove that $\mathcal{L}(\text{M2-rec}(\mathcal{R}, \varnothing)) = \mathcal{M}_2(\mathcal{R})$.

First, we introduce some preliminary definitions, and then we formalize the correctness theorem for M2-rec. Then, we proceed to prove by induction that

M2-rec is correct. Finally, we observe that the correctness of M2 is a corollary of the correctness of M2-rec.

## Preliminary Definitions

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 $\mathrm{L} : \wp(\mathbb{R}^{\mathcal{T}}) \rightarrow \mathbb{R}^{\mathcal{T}}$ 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 $i \in [1 \ldots n]$, $\mathcal{R}_i$ is not a concatenation. We define a function to determine whether a regex is a suffix of another modulo $\overline{*}$. $\mathrm{suff} : \mathbb{R}^{\mathcal{T}} \times \mathbb{R}^{\mathcal{T}} \rightarrow \mathrm{bool}$ is defined as $\mathrm{suff}(\mathcal{R}', \mathcal{R}_1 \cdots \mathcal{R}_n) = \mathrm{true}$ iff $\exists j \in [1 \ldots n]$ such that $r(\mathcal{R}') = r(\mathcal{R}_{j+1} \ldots \mathcal{R}_n)$. 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 $\{\, (a^*)^*, a^*(a^*)^{\overline{*}} \,\}$.

Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}$. If $\mathcal{R}_1 = \mathcal{R}_2$, then we define $\mathcal{M}_2^{\mathcal{R}_2} : \mathbb{R} \rightarrow \Sigma^*$ 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.

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

The words in $\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1)$ are those that can reach $\mathcal{R}_2$ in at least two different traces from $\mathcal{R}_1$, and then can be matched from $\mathcal{R}_2$.

## Correctness of M2-rec

We now formalize the correctness of M2-rec. We define a precondition for M2-rec, and then we give a postcondition. The correctness theorem, namely Theorem 3, states that the precondition implies the postcondition. Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}, E \in \wp(\mathbb{R}^{\mathcal{T}})$.

**Precondition**

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 $\operatorname{suff}(\mathcal{R}_i, \mathcal{R})$.

The precondition asserts that the actual arguments of the calls to M2-rec are consistent: it forbids calling the function with an arbitrary set of regexes as argument $E$. In particular, the second and the third conditions together ensure that $E$ is obtained by expanding the stars in $\mathcal{R}$. This is always verified if M2-rec is initially invoked with $E$ set to $\varnothing$. Observe that the precondition trivially holds for M2-rec$(\mathcal{R}, \varnothing)$ if $\mathcal{R} \in \mathbb{R}$. Furthermore, if $\mathcal{R} \in E$, then $\mathcal{R} = \mathrm{L}(E)$.

**Postcondition**

- 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 first case in the postcondition specifies that if $E$ is empty, then the language recognized by M2-rec$(\mathcal{R}, E)$ is exactly $\mathcal{M}_2(\mathcal{R})$. The second condition is more interesting, as it corresponds to the case in which $E$ is not empty, namely the algorithm is expanding 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 $\mathrm{L}(E)$, which is the star that the algorithm is expanding.

**Theorem 3** (Correctness of M2-rec)**.** *If the precondition holds for* M2-rec*, then the postcondition holds.*

Theorem 3 formalizes that if M2-rec is called with correct parameters, then it computes $\mathcal{M}_2$. In case the algorithm is expanding a star (that is, $E \neq \varnothing$), it computes the language of words that have a nonempty prefix that is matched in at least two different traces and can then reach the star.

Observe that if $\mathcal{R} \in \mathbb{R}$, the precondition holds for M2-rec$(\mathcal{R}, \varnothing)$. This implies that we can apply the correctness theorem and obtain that (by the second case in the postcondition) $\mathcal{L}(\text{M2-rec}(\mathcal{R}, \varnothing)) = \mathcal{M}_2(\mathcal{R})$. As mentioned at the beginning of this Section, this is equivalent to $\mathcal{L}(\text{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$, which is the statement of Theorem 1. In Corollary 1 we formalize that the correctness of M2 is a corollary of Theorem 3.

## Proof Structure

We prove that the precondition implies the postcondition by induction on the set of actual arguments that will be used in the subcalls of M2-rec. First, we formally define this set. 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 (that is, $\mathcal{R}_1$ has not been expanded yet). $\mathcal{A}(\mathcal{R}, E)$ is the set of actual arguments that will be used in the subcalls of M2-rec$(\mathcal{R}, E)$. 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 $\mathcal{R}_1 \in \mathrm{rch}(\mathcal{R})$ (since the algorithm explores all regexes that can be expanded during the concrete execution), and $\mathrm{rch}(\mathcal{R})$ 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 and show that the induction is well-founded.

*Proof.* We prove by induction on $\mathcal{A}(\mathcal{R}, E)$ that the precondition always implies the postcondition. The proof follows the same steps that appear in the sketch proof of Theorem 1 (see page 15).

## Base Case $(\mathcal{A}(\mathcal{R}, E) = \varnothing)$

If $\mathcal{A}(\mathcal{R}, E) = \varnothing$, then there are no subcalls to M2-rec. There are only three possible cases.

1. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle \varepsilon, \varepsilon \rangle$. Then, the execution reaches line 8 and $\perp$ is correctly returned, since no word in $\Sigma^+$ is matched in two different traces from $\varepsilon$.

2. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle \mathcal{R}_1^{\overline{*}}, \mathcal{R}_2 \rangle$. 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.

## Inductive Case $(\mathcal{A}(\mathcal{R}, E) \neq \varnothing)$, $E = \varnothing$

If $\mathcal{A}(\mathcal{R}, E) \neq \varnothing$, then we are in the inductive case and there are subcalls to M2-rec. We first consider the subcase in which $E = \varnothing$, that is the algorithm is not expanding any star. We show that $\mathcal{L}(\text{M2-rec}(\mathcal{R}, E)) = \mathcal{M}_2(\mathcal{R})$ in three different cases that depend on $\mathcal{R}$.

1. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R})\rangle = \langle a, \mathcal{R}_1\rangle$. 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 the call 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. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R})\rangle = \langle \mathcal{R}_1|\mathcal{R}_2, \mathcal{R}_3\rangle$. 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) the 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) the 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) the 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: $\mathcal{L}$(M2-rec$(\mathcal{R}_1\mathcal{R}_3, \varnothing)$) equals $\mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3)$ and $\mathcal{L}$(M2-rec$(\mathcal{R}_2\mathcal{R}_3, \varnothing)$) equals $\mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3)$. Observing that we return the regular expression $(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup^r$ M2-rec$(\mathcal{R}_1\mathcal{R}_2, \varnothing) \cup^r$ 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. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R})\rangle = \langle \mathcal{R}_1^*, \mathcal{R}_2\rangle$. 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 $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^r_{\notin} \mathcal{R}_2) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r \mathcal{R}_1^* \cdot \text{M2-rec}(\mathcal{R}_2, \varnothing)$, we can conclude:

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

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

## Inductive Case ($\mathcal{A}(\mathcal{R}, E) \neq \varnothing$), $E \neq \varnothing$

We now consider the other subcase in the inductive case: $E \neq \varnothing$. In this case, the algorithm is expanding a star, and we show that $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}) \subseteq \mathcal{L}(\text{M2-rec}(\mathcal{R}, E)) \subseteq \mathcal{M}_2(\mathcal{R})$. We prove this in three different cases that depend on $\mathcal{R}$.

1. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R}) \rangle = \langle a, \mathcal{R}_1 \rangle$. In this case we return $a \cdot \text{M2-rec}(r(\mathcal{R}_1), E)$. 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:

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

2. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R})\rangle = \langle \mathcal{R}_1|\mathcal{R}_2, \mathcal{R}_3\rangle$. 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 $(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_3, E) \cup^r \text{M2-rec}(\mathcal{R}_2\mathcal{R}_3, E)$, we obtain:

$$
\begin{aligned}
&\mathcal{M}_2^{\mathrm{L}(E)}((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3) \\
&= I \cup \mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2\mathcal{R}_3) \\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_3, E) \cup^r \text{M2-rec}(\mathcal{R}_2\mathcal{R}_3, E)) \\
&\qquad\qquad\qquad\qquad \text{(inductive hypothesis and } I \subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3)) \\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap^r_{\notin} \mathcal{R}_2\mathcal{R}_3)) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3) \qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2((\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3) \qquad\qquad \text{(analogous to subcase } (\mathcal{R}_1|\mathcal{R}_2)\mathcal{R}_3 \text{ if } E = \varnothing)
\end{aligned}
$$

3. $\langle \mathrm{hd}(\mathcal{R}), \mathrm{tl}(\mathcal{R})\rangle = \langle \mathcal{R}_1^*, \mathcal{R}_2\rangle$. 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 the language $\mathcal{M}_2^{\mathrm{L}(E\cup\{\mathcal{R}_1^*\mathcal{R}_2\})}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$; (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 we obtain 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 $\mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^{r}_{\notin} \mathcal{R}_2)$. 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 $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^{r}_{\notin} \mathcal{R}_2) \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r \mathcal{R}_1^* \cdot \text{M2-rec}(\mathcal{R}_2, E)$, we obtain:

$$
\begin{aligned}
&\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1^*\mathcal{R}_2)\\
&= I \cup \mathcal{M}_2^{\mathrm{L}(E \cup \{\mathcal{R}_1^*\mathcal{R}_2\})}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2)\\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^{r}_{\notin} \mathcal{R}_2)\\
&\qquad \cup^r \text{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \cup^r \mathcal{R}_1^* \cdot \text{M2-rec}(\mathcal{R}_2, E))\\
&\hfill \text{(inductive hypothesis and } I \subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^{r}_{\notin} \mathcal{R}_2))\\
&\subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap^{r}_{\notin} \mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_2) \qquad \text{(inductive hypothesis)}\\
&= \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2) \qquad \text{(analogous to subcase } \mathcal{R} = \mathcal{R}_1^*\mathcal{R}_2 \text{ if } E = \varnothing)
\end{aligned}
$$

□

The overall correctness of M2 (Theorem 1) is then a corollary of the correctness of M2-rec (Theorem 3).

**Corollary 1** (Correctness of M2)**.** *Let* $\mathcal{R} \in \mathbb{R}$.

$$\mathcal{L}(\text{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$$

*Proof.* Follows immediately from the fact that $\text{M2}(\mathcal{R})$ is $\text{M2-rec}(\mathcal{R}, \varnothing)$. The precondition of Theorem 3 holds for the arguments. By applying Theorem 3, we can observe what follows.

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

□