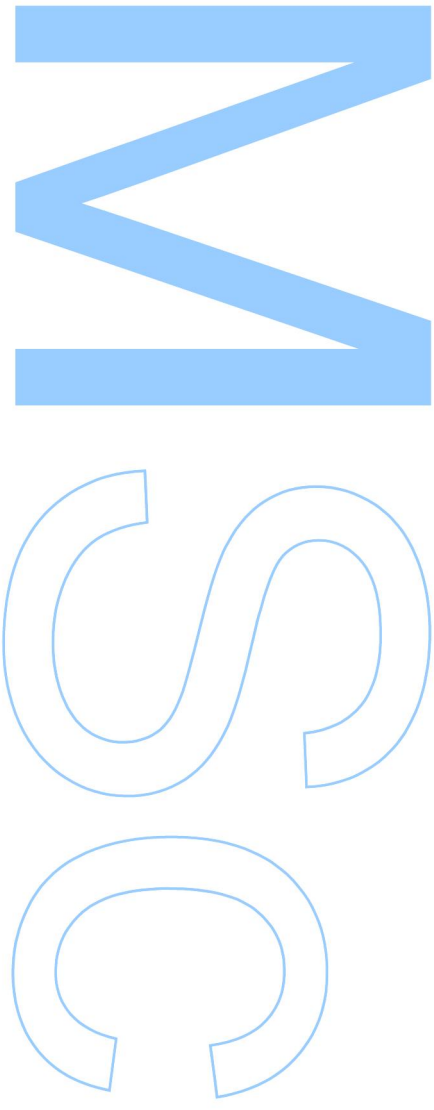
# Efficient Regular Pattern Matching avoiding Denial of Service



Daniel Afonso de Resende
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2025

**Orientador**
Nelma Resende Araújo Moreira, Categoria
Faculdade de Ciências da Universidade do Porto

**Coorientador**
Nome do Orientador, Categoria
Faculdade de Ciências da Universidade do Porto


Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

# Abstract

Regular expressions (*regex*) are a foundational tool in modern software, widely used for string matching, input validation, and text parsing. Despite their utility, improperly constructed regular expressions can introduce serious security vulnerabilities. One of the most critical threats is the Regular Expression Denial of Service (*ReDoS*) attack, in which carefully crafted inputs cause the regex engine to perform excessive and redundant processing. This results in dramatic slowdowns or even complete unresponsiveness of the system. ReDoS poses a significant risk to web applications, APIs, and other input-facing systems, where user-controlled input is matched against vulnerable patterns.

In this work, we propose a system to address ReDoS by transforming regular expressions into a modified *position automata*, a form of nondeterministic finite automaton (NFA) that tracks the exact start and end positions of all matches within an input string. This structure enables a matching function that computes *all* match positions, including overlapping ones, without relying on backtracking. By exhaustively and efficiently exploring the automaton's transitions, our approach avoids the exponential blowup typical of vulnerable engines, while preserving a somewhat full regex expressiveness.

Furthermore, we also review and compare this approach with existing solutions present in state-of-the-art programming languages and libraries, such as *RE#*. **Palavras-chave:** regular expressions, ReDoS, position automata, nondeterministic finite automata, pattern matching.

# Resumo

O teu resumo COOL, its me TEST WOWIEESSS

**Palavras-chave:** palavra, chave..

# Acknowledgements

First of all, I would like to thank my family, etc, etc

**Dedico à minha mãe . . .**

# Contents

# List of Figures

# Listings

# Acronyms

# Chapter 1

# Introduction

In this chapter, the problem is overviewed, the study's importance is explained along with goals for the proposed solution.

## 1.1 Background

# Chapter 2

# Preliminaries

Theory builds upon theory, therefore it is essential to establish a solid foundation by understanding the basic concepts and terminology that compose the core topics of formal languages and automata theory. In this chapter we begin by formally defining what a language is and then move on to describe the class of languages known as regular languages. Along the way, we will also introduce various concepts such as finite/non-finite automata and regular expressions.

## 2.1 Alphabets, Strings and Languages

### Alphabets

An *alphabet* is a finite, non-empty set of symbols, typically denoted by the Greek letter $\Sigma$. That is,

$$\Sigma = \{a_1, a_2, \ldots, a_n\}$$

where each $a_i$ is a symbol in the alphabet.

For example, one can represent the binary alphabet as $\Sigma = \{0, 1\}$, or the English alphabet as $\Sigma = \{a, b, c, \ldots, z\}$.

### Strings

A *string* over an alphabet $\Sigma$ is a finite sequence of symbols from $\Sigma$. Strings are typically denoted by $w$, and the *length* of a string $w$ is denoted by $|w|$.

The set of all strings over the alphabet $\Sigma$ is denoted by $\Sigma^*$ and defined as:

$$\Sigma^* = \{w \mid w \text{ is a finite sequence of symbols from } \Sigma\}$$

The unique string of length zero is called the *empty string*, denoted by $\varepsilon$. It is important to note that $\varepsilon \in \Sigma^*$.

For example, if $\Sigma = \{0, 1\}$, then we have that:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}$$

Where the empty string is, as mentioned above, denoted by $\varepsilon$ and also belongs to $\Sigma^*$.

### Languages

A *language* over an alphabet $\Sigma$ is a set of strings over $\Sigma$.

$$L \subseteq \Sigma^*$$

That is, a language is any subset of $\Sigma^*$, possibly infinite, finite, or even empty.
Since a language is a set of strings, the following standard set operations can be applied (assuming $A$ and $B$ are languages over the same alphabet $\Sigma$):

- *Intersection*: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- *Union*: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- *Difference*: $A - B = \{x \mid x \in A \text{ and } x \notin B\}$

Furthermore, we can also operate specifically over languages with the following operations (assuming $L_1$ and $L_2$ are languages over the same alphabet $\Sigma$):

- *Concatenation*: $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- *Kleene Star*: $L^* = \bigcup_{n=0}^{\infty} L^n$, where $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$.
- *Reversal*: $L^R = \{x^R \mid x \in L\}$, where $x^R$ denotes the reversal of string $x$.
- *Complement*: $\overline{L} = \Sigma^* - L$, i.e., the set of all strings over $\Sigma$ that are not in $L$.
- *Intersection with regular sets*: Often used to verify whether a language satisfies certain regular constraints, since the class of regular languages is closed under intersection.

These operations form the basis for reasoning about the expressiveness and closure properties of language classes such as regular, context-free, and context-sensitive languages. In particular, regular languages are closed under all the operations listed above, including union, intersection, concatenation, and Kleene star. This robustness makes them especially amenable to algorithmic manipulation, as seen in finite automata and regular expression engines.

## 2.2 Regular Expressions

Let $\Sigma$ be a finite alphabet. Let $L \subseteq \Sigma$. The set of *regular expressions* over $\Sigma$, denoted by RegExp($\Sigma$), is defined inductively as follows:

- $\emptyset$ is a regular expression denoting the empty language: $L(\emptyset) = \emptyset$.
- $\varepsilon$ is a regular expression denoting the language containing only the empty string: $L(\varepsilon) = \{\varepsilon\}$.
- For each symbol $a \in \Sigma$, $a$ is a regular expression denoting the singleton language: $L(a) = \{a\}$.
- If $r_1$ and $r_2$ are regular expressions, then so are:
  - $(r_1 \mid r_2)$, denoting the union: $L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$.
  - $(r_1 \cdot r_2)$, denoting concatenation: $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$.
  - $(r_1)^*$, denoting Kleene star: $L(r_1^*) = (L(r_1))^*$.

We write RegExp($\Sigma$) to denote the set of all such syntactic expressions, and for each $r \in$ RegExp($\Sigma$), the function $L(r)$ yields the language defined by $r$.

Parentheses are used to disambiguate expressions and enforce precedence; by convention, Kleene star binds most tightly, followed by concatenation, and finally union.

### 2.2.1 Extended Regular Expressions

In addition to the basic operations, some extended operators are often used for convenience. These include:

- **Kleene plus**: Given a regular expression $r$, the expression $r^+$ denotes one or more repetitions of $r$:
$$L(r^+) = L(r) \cdot L(r)^*.$$
- **Fixed repetition (power)**: For a regular expression $r$ and integer $n \geq 0$, the expression $r^n$ denotes $n$ consecutive concatenations of $r$:
$$L(r^0) = \{\varepsilon\}, \quad L(r^n) = L(r) \cdot L(r^{n-1}) \text{ for } n > 0.$$
- **Bounded repetition**: For a regular expression $r$ and integers $m, n$ with $0 \leq m < n$, the bounded repetition $r^{[m,n[}$ denotes the language containing all strings formed by concatenating between $m$ and $n$ copies of strings from $L(r)$:
$$L(r^{[m,n[}) = \bigcup_{k=m}^{n} L(r^k).$$

These extended forms do not increase the expressive power of regular expressions but are useful for readability and practical applications. They can always be rewritten using the fundamental operators: union and concatenation.

### 2.2.2 Finite Automata

A *finite automaton* is a theoretical machine used to recognize regular languages. It processes input strings symbol by symbol and determines whether the string belongs to the language defined by the automaton. There are two main types of finite automata:

- **Deterministic Finite Automaton (DFA)**: An automaton where, for each state and input symbol, there is exactly one possible next state.
- **Nondeterministic Finite Automaton (NFA)**: An automaton that allows multiple possible transitions for a given state and input symbol, including transitions without consuming any input (called $\varepsilon$-transitions).

#### Formal Definition of an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- $Q$ is a finite set of states,
- $\Sigma$ is the input alphabet,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting (final) states.

A string $w \in \Sigma^*$ is accepted by the NFA if there exists a sequence of transitions (possibly including $\varepsilon$-moves) that consumes $w$ and ends in a state $q$ such that $q \in F$.

### 2.2.3 Position Automata

To first recall the notion of *position automata* (also known as *Glushkov automata*), one must first understand the idea of marking each occurrence of a symbol in a regular expression with a unique position. This transforms a regular expression into a *marked regular expression*, where each letter is indexed according to its position in the expression when read left to right. The position automaton is then constructed from this marked version by associating each position

with a distinct state, and defining transitions based on the structural analysis of the expression. [1]

The key to this construction lies in the inductive computation of three sets: *First*, *Last*, and *Follow*.The *First* set contains positions that may begin a word in the language; the *Last* set includes positions that may end such words; and the *Follow* relation determines which positions can immediately follow a given position in any word generated by the expression. The resulting automaton has states corresponding to positions, transitions derived from the *Follow* relation, an initial state representing position 0 (the start of matching), and final states determined by the *Last* set, extended with position 0 if the empty string is accepted. [1]

### 2.2.4 Derivatives

The *derivative of a regular expression* was first introduced in 1962 by Janusz Brzozowski. It is a powerful concept used to define the behavior of regular expressions in a more operational manner. The derivative of a regular expression $r$ with respect to a symbol $a$ is another regular expression $D_a(r)$ that describes the set of strings that can be obtained by taking the derivative of $r$ with respect to $a$.

The derivative can be defined based on the structure of the regular expression:

- If $r = \varepsilon$, then $D_a(r) = \emptyset$ for all $a \in \Sigma$.
- If $r = a$, then $D_a(r) = \varepsilon$.
- If $r = r_1 \cdot r_2$, then $D_a(r) = D_a(r_1) \cdot L(r_2) \cup \varepsilon \cdot D_a(r_2)$.
- If $r = r_1 + r_2$, then $D_a(r) = D_a(r_1) + D_a(r_2)$.
- If $r = r^*$, then $D_a(r) = D_a(r) \cdot r^*$.

# Chapter 3

# State of the Art

## 3.1 Overview of XYZ

Computers are devices that

# Chapter 4

# Development

# Chapter 5

# Matching

Regex *matching* is the process of checking whether a piece of text fits a specific pattern described using a regular expression (regex). A regex is a compact, rule-based way to describe sets of strings—like email addresses, phone numbers, or specific word formats.

In this chapter, we will discuss the different approaches to matching regular expressions, the implications of using them, and the performance considerations that arise from these choices. Furthermore, we will also present a novel approach based on position automata, which aims to mitigate the performance issues associated with traditional regex engines while preserving some of the extended expressiveness of regex patterns.

## 5.1 Modified Position Automata

A *position automaton* is a type of nondeterministic finite automaton (NFA) that facilitates overlapped matching.

**Algorithm 1** NFAPOSCOUNT($R$): Construct Special Position Automaton

**Require:** Regular expression $R$
**Ensure:** A NFA $A$

$A \leftarrow$ new empty NFA
$i \leftarrow A$.addState()
$A$.addInitial($i$)
**if** $R.\Sigma \neq \emptyset$ **then**
    $A$.setSigma($R.\Sigma$)
**end if**
$A$.addTransitionStar($i, i$) ▷ Enable overlapping matches
$f_R \leftarrow R$.marked()
stack $\leftarrow$ empty stack
addedStates $\leftarrow$ empty map
**for all** $p \in f_R$.First() **do**
    $q \leftarrow A$.addState($p$)
    addedStates[$p$] $\leftarrow q$
    stack.push($(p, q)$)
    $A$.addTransition($i, p$.symbol(), $q$)
**end for**
FollowSets $\leftarrow f_R$.followListsD()
**while** stack is not empty **do**
    $(s, s_{\text{idx}}) \leftarrow$ stack.pop()
    **for all** $t \in$ FollowSets[$s$] **do**
        **if** $t \in$ addedStates **then**
            $q \leftarrow$ addedStates[$t$]
        **else**
            $q \leftarrow A$.addState($t$)
            addedStates[$t$] $\leftarrow q$
            stack.push($(t, q)$)
        **end if**
        $A$.addTransition($s_{\text{idx}}, t$.symbol(), $q$)
    **end for**
**end while**
$e \leftarrow A$.addState()
$A$.addTransitionStar($e, e$)
**for all** $p \in f_R$.Last() **do**
    **if** $p \in$ addedStates **then**
        $A$.addFinal(addedStates[$p$])
        $A$.addTransitionStar(addedStates[$p$], $e$)
    **end if**
**end for**

## 5.2 Automata-Based Matching

Given a regular expression $R$, one can construct an NFA $A$ such that $L(A) = L(R)$. Matching then reduces to verifying whether the automaton $A$ accepts the input string $s$. In DFA-based engines, each character of the input leads to a deterministic transition from one state to another, resulting in a guaranteed linear-time match. In contrast, NFA-based engines may involve branching paths due to nondeterminism and can require simulating multiple transitions concurrently.

For example, consider the following regex pattern:

```
^(a+)+$
```

## 5.3 Matching with the Modified Position Automaton

One can find all matches over an input string by constructing the modified position automaton from a regular expression and then simulating the automaton's transitions over the input string. The algorithm presented in this section is designed to track the start and end positions of all matches, including overlapping ones, without relying on backtracking.

**Algorithm 2** TABLEMATCHER$(A, s)$: Modified Position Automaton Multi-matcher

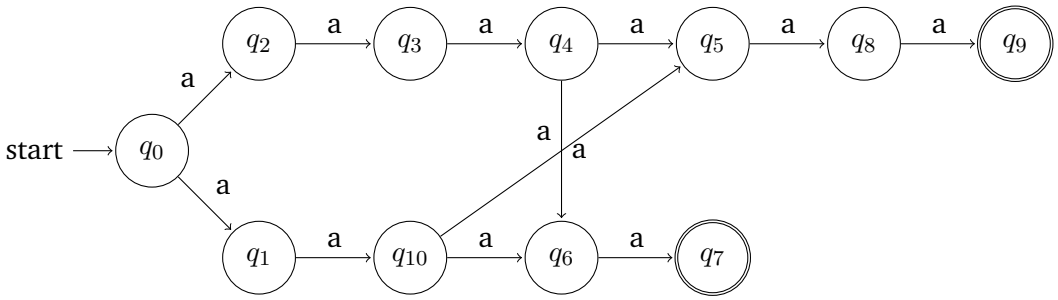
**Require:** $A = (\Sigma, Q, \delta, I, F)$: NFA
**Require:** $s$: input string
**Ensure:** $M$: mapping from final states to lists of match positions

$symbols \leftarrow$ list with $\varepsilon$ prepended to $s$
$currentRow \leftarrow$ empty map from states to list of position pairs
$finalMatches \leftarrow$ empty map from states to list of matches
$position \leftarrow 0$
**for all** $sym$ in $symbols$ **do**
    **if** $sym = \varepsilon$ **then**
        **for all** $q_0 \in I$ **do**
            $currentRow[q_0] \leftarrow [(0, 0)]$
        **end for**
    **else**
        $nextRow \leftarrow$ empty map
        **if** $sym \in \Sigma$ **then**
            **for all** $q \in$ **keys**$(currentRow)$ **do**
                $transitions \leftarrow \delta(q, sym)$
                **if** $transitions$ is not empty **then**
                    **for all** $q' \in transitions$ **do**
                        **for all** $(start, \_) \in currentRow[q]$ **do**
                            **if** $q' = q$ and $q' \in I$ **then**
                                append $(position, position)$ to $nextRow[q']$
                            **else**
                                append $(start, position)$ to $nextRow[q']$
                                **if** $q' \in F$ **then**
                                    append $(start, position)$ to $finalMatches[q']$
                                **end if**
                            **end if**
                        **end for**
                    **end for**
                **end if**
            **end for**
        **else** ▷ Symbol not in $\Sigma$; treat as fresh start
            **for all** $q_0 \in I$ **do**
                $nextRow[q_0] \leftarrow [(position, position)]$
            **end for**
        **end if**
        $currentRow \leftarrow nextRow$
        $position \leftarrow position + 1$
    **end if**
**end for**
**return** $finalMatches$

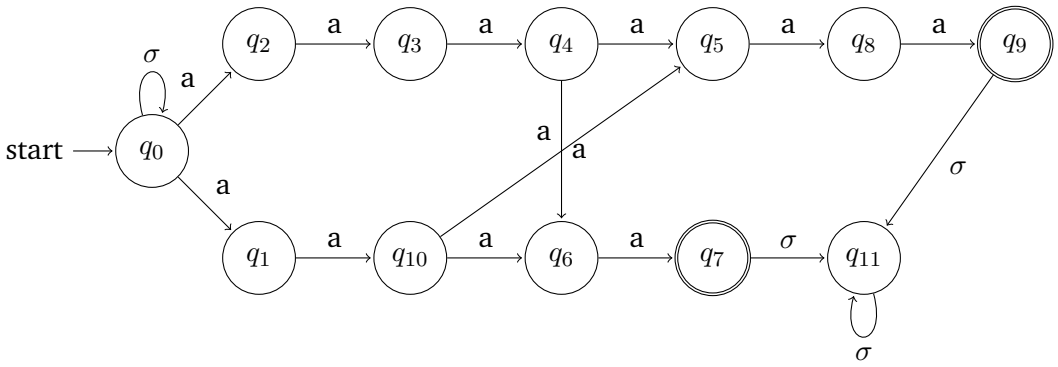
As an example, consider the following regular expression $R$ and input string $w$:

$$R = (aa + aaa)(aaa + aa)$$
$$w = aaaaabaaaaa$$

The Glushkov automaton construction for $R$ is as follows:




Meanwhile, the modified position automaton for this regular expression can be constructed using the algorithm presented in 1, resulting in the following:




One can then apply the matching algorithm (2) to an input string, resulting in the following match table:

# Chapter 6

# Results and Discussion

This is a test

## 6.1 Algorithm Analysis

## 6.2 Accuracy

The methods of evaluating

## 6.3 Comparison with Other Methods

## 6.4 Examples

# Chapter 7

# Conclusion

## 7.1 Findings Summary

This research and development project served the objective of

## 7.2 Contributions

# Bibliography

[1] S Broda et al. 'A mesh of automata'. In: *Information and Computation* 265 (2019), pp. 94–111. DOI: 10.1016/j.ic.2019.01.003 (cit. on p. 7).