# Static Analysis for Regular Expression Exponential Runtime via Substructural Logics

Asiri Rathnayake[1] and Hayo Thielecke[1]

University of Birmingham, Birmingham B15 2TT, United Kingdom
`h.thielecke@cs.bham.ac.uk`

**Abstract.** Regular expression matching using backtracking can have exponential runtime, leading to an algorithmic complexity attack known as REDoS in the systems security literature. In this paper, we present a static analysis that detects whether a given regular expression can have exponential runtime for some inputs. The analysis works by forming powers and products of transition relations and thereby reducing the REDoS problem to reachability. The correctness of the analysis is proved using a substructural calculus of search trees, where the branching of the tree causing exponential blowup is characterized as a form of non-linearity.

## 1 Introduction

Regular expressions are everywhere. Yet the backtracking virtual machines that are used to match them (in Java, .NET and other frameworks) are very different from the DFA construction used in compiling. Whereas DFAs run in linear time but may be expensive to construct, backtracking matchers have low initial cost, but may have exponential runtime for some inputs [6]. This is a problem when such matchers may be exposed to malicious input, say over a network, as an attacker could craft an input in order for the matcher to take exponential time. This problem is known as REDoS, short for Regular Expression Denial-of-Service.

For a straightforward example of an exponential blowup, consider the following regular expression:

$$(a \mid b \mid ab)^* c$$

Matching this expression against input strings of the form $(ab)^n$ leads the Java virtual machine to a halt for very moderate values of $n$ ($\sim 50$) on a contemporary computer. Certain other backtracking matchers like the PCRE library and the matcher available in the .NET platform seem to handle this particular example well. However, the ad-hoc nature of the workarounds implemented in these frameworks are easily exposed with a slightly complicated expression / input combination:

$$(a \mid b \mid ab)^* bc$$

This expression, when matched against input strings of the form $(ab)^n ac$, leads to exponential blowups on all the three matchers mentioned.

The REDoS analysis builds on the idea of non-deterministic Kleene expressions. When matching the input string $ab$ against the Kleene expression $(a \mid b \mid ab)^*$, a match could be found by taking either of the two different paths through the corresponding NFA. If we repeat this string to form $abab$, now there are four different paths through the NFA; this process quickly builds up to an exponential amount of paths through the NFA as the pumpable string $ab$ is repeated. A matcher based on DFAs would not face a difficulty in dealing with such expressions since the DFA construction eliminates such redundant paths. However, these expressions can be fatal for backtracking matchers based on NFAs, as their operation depends on performing a depth-first traversal of the entire search space.

We think of the various phases of the analysis as very simple and non-standard logics for judgements for different implications of the form:

$$w : p_1 \rightarrow p_2$$

Here $p_1 \rightarrow p_2$ is a proposition and $w$ is a proof of it, which we will call its realizer. In this way, we can focus on what the analysis tries to construct, not how. Hence the analysis can be seen as a form of proof search, and it is implemented via straightforward closure algorithms.

A second use of logic or type theory in this paper comes in when proving the soundness of the analysis, when we need to show that the constructed string really leads to exponential runtime. While the backtracking machine that we use as an idealization of backtracking matchers (like those in the Java platform) is not very complicated, it is not straightforward to reason about how it behaves on some constructed malicious input. This is because the machine traverses the search tree in a depth-first strategy, whereas the attack string is best understood in terms of a composition of horizontal slices of the search tree. To reason compositionally, we first introduce a calculus of search trees, inspired by substructural logics. In a nutshell, the existence of a pumpable string as part of a REDoS vulnerability amounts to the existence of a non-linear derivation in the search tree logic, essentially as in a derivation of this form:

$$\frac{p}{p, p}$$

Thus we can reason about the exponential growth of the search tree in a compositional, logical style, separate of the search strategy of the backtracking matcher. The exponential runtime of the machine then follows due to the fact that the runtime is at least the width of the search tree if the matcher is forced to explore the whole tree.

## Outline of the paper

Section 2 presents some required background on regular expression matching in a form that will be convenient for our purpose. We then define the three phases (prefix, pumping, and suffix construction) of our REDoS analysis in Section 3

| | |
|---|---|
| $a$, $b$, $c$ | input symbols |
| $w$, $x$, $y$, $z$ | strings of input symbols |
| $p, q$ | pointers to NFA nodes or states |
| $\beta$, $\theta$ | ordered sequences $p_1 \ldots p_n$ of NFA states |
| $\Phi$ | sets $\{p_1, \ldots, p_n\}$ of NFA states |
| $\sigma$ | sequences of state/index pairs $(p, j)$ |
| $\varepsilon$ | empty word or sequence |
| $\twoheadrightarrow$ | NFA transition relation (Definition 2) |
| $\mapsto$ | ordered NFA transition function (Definition 1) |
| $\rightsquigarrow$ | transitions of backtracking machine (Definition 3) |
| $\Rightarrow$ | multistate transition function (Definition 5) |
| $\vec{\Gamma}_\ell$ | ordered multistate transition function (Definition 8) |

Fig. 1: Notational conventions

and validate it on some examples in Section 4. Section 5 and Section 6 prove the soundness and the completeness of the analysis using a substructural calculus of search trees. Section 7 presents a brief overview of the OCaml implementation of the analysis and the practical performance of our tool. We conclude with a discussion of related work in Section 8 and directions for further work in Section 9.

## 2 Basic constructs

This section presents some background material that will be needed for the analysis, such as non-deterministic automata. Figure 1 gives an overview of notation. We assume that the regular expression has been converted into an automaton following one of the standard constructions.

### 2.1 Backtracking and the ordered NFA

The usual text-book definitions of NFAs do not impose any ordering on the transition function. For an example, a traditional NFA for the regular expression $a(bc \mid bd)$ would not prioritize any of the two transitions available for character $b$ over the other. Since backtracking matchers follow a greedy left-to-right evaluation of alternations, the alternation operator effectively becomes non-commutative in their semantics for regular expressions. Capturing this aspect in the analysis requires a specialized definition of NFAs.

If we are only concerned about acceptance, Kleene star is idempotent and alternation is commutative. If we are interested in exponential runtime, they are not. The non-commutativity of alternation is not that surprising in terms of programming language semantics, as Boolean operators like `&&` in C or `andalso` in ML have a similar semantics: first the left alternative is evaluated, and if that does not evaluate to true, the right alternative is evaluated. Since in our tool the

NFA is constructed from the syntax tree, the order is already available in the data structures. The children of a NFA node have a left-to-right ordering.

**Definition 1 (Ordered NFA).** *An ordered NFA $\mathcal{N}$ consists of a set of states, an initial state $p_0$, a set of accepting states* **Acc** *and for each input symbol $a$ a transition function from states to sequences of states. We write this function as*

$$a : p \mapsto q_1 \ldots q_n$$

*For each input symbol $a$ and current NFA state $p$, we have a sequence of successor states $q_i$. The order is significant, as it determines the order of backtracking.*

In the textbook definition of an $\varepsilon$-free NFA, the NFA has a transition function $\delta$ of type

$$\delta : (Q \times \Sigma) \to 2^Q$$

where $Q$ is the set of states and $\Sigma$ the set of input symbols. Here we have imposed an order on the sets in the image of the function, replacing $2^Q$ by $Q^*$, curried the function, and swapped the order of $Q$ and $\Sigma$.

$$\Sigma \to (Q \to \quad Q^*)$$
$$a \mapsto \quad p \ \mapsto q_1 \ldots q_n$$

**Definition 2.** *The nondeterministic transition relation of the NFA is given by the following inference:*

$$\frac{a : p \mapsto q_1 \ldots q_n}{a : p \twoheadrightarrow q_i}$$

Note however, that we cannot recover the ordering of the successor states $q_i$ from the non-deterministic transition relation. In this regard, the NFA on which the matcher is based has a little extra structure compared to the standard definition of NFA in automata theory. If we know that

$$a : p \twoheadrightarrow q_1 \text{ and } a : p \twoheadrightarrow q_2$$

we cannot decide whether the ordered transition is

$$a : p \mapsto q_1 q_2 \text{ or } a : p \mapsto q_2 q_1$$

To complement the ordered NFA, we introduce two kinds of data structures: ordered multistates $\beta$ are finite sequences of NFA states $p$, where the order is significant. Multistates $\Phi$ represent sets of NFA states, so they can be represented as lists, but are identified up to reordering. Each ordered multistate $\beta$ can be turned into a multistate given by the set of its elements. We write this set as $\mathsf{Set}(\beta)$. If

$$\beta = p_1 \ldots p_n$$

then

$$\mathsf{Set}(\beta) = \{p_1, \ldots, p_n\}$$

The difference between $\beta$ and $\mathsf{Set}(\beta)$ may appear small, but the notion of equality for sets is less fine-grained than for sequences, which has an impact on the search space that the analysis has to explore.

## 2.2 The abstract machines

The analysis assumes exact matching semantics of regular expressions. Given regular expression $e$ and the input string $w$, the matcher is required to find a match of the entire string, as opposed to a sub-string. Most practical matchers search for a sub-match by default. However, such behavior can be modeled in exact matching semantics by augmenting the regular expression with match-all constructs at either end of the expression, as in $(.^*e.^*)$. Practical implementations offer special "anchoring" constructs that allow regular expression authors to enforce exact matching semantics. For an example, expressions of the form $(\char`^e\$)$ require them to be matched against the entire input string.

While the theoretical formulation of our analysis assumes exact matching semantics (thus avoiding unnecessary clutter), our implementation assumes sub-match semantics, since it is more useful in practice. The translation between the two semantics is quite straightforward.

**Definition 3 (Backtracking abstract machine).** *Given an ordered NFA, the backtracking machine is defined as follows. We assume an input string $w$ as given. Machine transitions may depend on $w$, but it does not change during transitions, so that we do not explicitly list it as part of the machine state. The input symbol at position $j$ in $w$ is written as $w[j]$ ($j$ is 0-based).*

- *States of the backtracking machine are finite sequences of the form*

$$\sigma = (p_0, j_0) \ldots (p_n, j_n)$$

  *where each of the $p_i$ is an NFA state and each of the $j_i$ is an index into the current input string. We refer to individual $(p, i)$ pairs as frames (as in stack frames).*
- *The initial state of the machine is the sequence of length 1 containing the frame:*
$$(p_0, 0)$$
- *The machine has matching transitions, which are inferred from the transition function of the ordered NFA as follows:*

$$\frac{w[j] = a \qquad a : p \mapsto q_1 \ldots q_n}{w \Vdash (p, j)\, \sigma \rightsquigarrow (q_1, j+1) \ldots (q_n, j+1)\, \sigma}$$

- *The machine has failing transition, of the form*

$$(p, j)\, \sigma \rightsquigarrow \sigma$$

  *where $w[j] \neq a$ or $j$ is the length of $w$ and $p \notin \mathbf{Acc}$.*
- *Accepting states are of the form:*

$$w \Vdash (p, j)\, \sigma$$

  *where $p \in \mathbf{Acc}$ and $j$ is the length of $w$.*

- *Transition sequences in $n$ steps are written as $\overset{n}{\rightsquigarrow}$ and inferred using the following rules:*

$$\frac{w \Vdash \sigma \rightsquigarrow \sigma'}{w \Vdash \sigma \overset{1}{\rightsquigarrow} \sigma'} \qquad \frac{w \Vdash \sigma_1 \overset{n}{\rightsquigarrow} \sigma_2 \qquad w \Vdash \sigma_2 \overset{m}{\rightsquigarrow} \sigma_3}{w \Vdash \sigma_1 \overset{n+m}{\rightsquigarrow} \sigma_3}$$

*We write $w \Vdash \sigma_1 \overset{*}{\rightsquigarrow} \sigma_2$ for $\exists n . w \Vdash (\sigma_1 \overset{n}{\rightsquigarrow} \sigma_2)$.*
- *Final states are either accepting or the empty sequence.*

The state of the backtracking machine is a stack that implements failure continuations. When the state is of the form $(p, j)\sigma$, the machine is currently trying to match the symbol at position $j$ in state $p$. Should this match fail, it will pop the stack and proceed with the failure continuation $\sigma$.

**Lemma 1.** *For any backtracking machine run:*

$$w \Vdash \sigma \overset{n}{\rightsquigarrow} \sigma'$$

*And for any $\bar{\sigma}$, the following run also exists:*

$$w \Vdash \sigma \, \bar{\sigma} \overset{n}{\rightsquigarrow} \sigma' \, \bar{\sigma}$$

*Proof.* Observe that each transition taken by the the first machine can be simulated on the extended machine. Moreover, each transition of the extended machine leaves the additional $\bar{\sigma}$ untouched.

The backtracking machine definition leaves a lot of leeway to the implementation. Implementation details are abstracted in the ordered transition relation. The most important choice in the definition is that the machine performs a depth-first traversal of the search tree. In principle, a backtracking matcher could also use breadth-first search. In that case, our REDoS analysis would not be applicable, and such matchers may avoid exponential run-time. However, the space requirements of breadth-first search are arguably prohibitive. A more credible alternative to backtracking matchers is Thompson's matcher [31,6,7], which is immune to REDoS. However the relative inflexibility of the lockstep algorithm (when supporting extended, non-regular pattern matching constructs) has made is less popular among practical regular expression libraries. The REDoS problem in the backtracking paradigm therefore remains quite significant.

**Definition 4 (Lockstep abstract machine).** *The lockstep abstract machine, based on Thompson's matcher [31], is defined as follows.*

- *The states of the lockstep matcher are of the form*

$$(\Phi, j)$$

*where $\Phi$ is a set of NFA states and $j$ is an index into the input string $w$.*
- *The intial state is*
$$(\{p_0\}, 0)$$

- *The matching transition are inferred as follows:*

$$\frac{w[j] = a \qquad a : p_1 \mapsto \beta_1 \quad \ldots \quad a : p_n \mapsto \beta_n}{(\{p_1, \ldots, p_n\}, j) \rightsquigarrow (\mathsf{Set}(\beta_1) \cup \ldots \cup \mathsf{Set}(\beta_n), j + 1)}$$

- *An accepting state is of the form*

$$(\Phi, j)$$

  *where $j$ is the length of $w$ and $\Phi \cap \mathbf{Acc} \neq \emptyset$.*

After each step, redundancy elimination is performed by taking sets rather than sequences.

The lockstep machine will not be used in the rest of the paper. It is only presented here to illustrate how it avoids the state-space explosion through redundancy elimination.

## 2.3   The power DFA construction

Based on a construction that is standard in automata theory and compiler construction, for each NFA there is a DFA. The set of states of this DFA is the powerset of the set of states of the NFA. We refer to such sets of NFA states as multistates.

**Definition 5 (Power DFA).** *Given an NFA, its power DFA is constructed as follows:*

- *The states of the power DFA are sets $\Phi$ of NFA states.*
- *The transition relation $\Rightarrow$ is defined as*

$$a : \Phi_1 \Rightarrow \Phi_2$$

  *if and only if*
$$\Phi_2 = \{p_2 \mid \exists p_1 \in \Phi_1.a : p_1 \twoheadrightarrow p_2\}$$

- *The initial state of the power DFA is the singleton set $\{p_0\}$.*
- *The accepting states of the power DFA are those sets $\Phi$ for which $\Phi \cap \mathbf{Acc} \neq \emptyset$.*

**Definition 6.** *The transition function of the power DFA is extended from strings $w$ to sets of strings $W$ using the following rule*

$$\frac{\Phi_2 = \{p_2 \mid \exists p_1 \in \Phi_1.\exists w \in W.w : p_1 \twoheadrightarrow p_2\}}{W : \Phi_1 \Rightarrow \Phi_2}$$

Intuitively, we regard $W$ as the set of realizers that take us from $\Phi_1$ to $\Phi_2$. Note that it is not only the case that (elements of) $W$ will take us from (elements of) $\Phi_1$ to (elements of) $\Phi_2$. Moreover, everything in $\Phi_2$ arises this way from $\Phi_1$ and $W$. In that sense, a judgement $W : \Phi_1 \Rightarrow \Phi_2$ is stronger than realizability or pre- and post-conditions. The fact that $\Phi_2$ is uniquely determined by $\Phi_1$ and $W$ is useful for the analysis.

# 3 The REDoS analysis

For a given regular expression of the form $e_1 e_2{}^* e_3$, the analysis attempts to derive an attack string of the form:

$$xy^n z$$

The presence of a pumpable string $y$ signals the analyser that a corresponding prefix $x$ and a suffix $z$ need to be derived in order to form the final attack string configuration. The requirements on the different segments of the attack string are as follows:

$$x : x \in L(e_1)$$
$$y : y \in L(e_2{}^*) \quad \text{(with } b > 1 \text{ paths)}$$
$$z : xy^n z \notin L(e_1 e_2{}^* e_3)$$

Intuitively, the prefix $x$ leads a backtracking matcher to a point where it has to match the (vulnerable) Kleene expression $e_2{}^*$. At this point the matcher is presented with $n$ $(n > 0)$ copies of the pumpable string $y$, increasing the search space of the matcher to the order of $b^n$. At the end of each of the search attempts (paths through the NFA), the suffix $z$ causes the matcher to backtrack, forcing an exploration of the entire search space.

## 3.1 The phases of the REDoS analysis

Overall, the REDoS analysis of a node $p_\ell$ (loop node) consists of three phases. The phases all work by incrementally exploring a transition relation. These relations are the power DFA transition relation $\Rightarrow$ and a new ordered variant $\vec{\Gamma}_\ell$ (Definition 8). The three analysis phases construct a REDoS prefix $x$, a pumpable string $y$ and a REDoS suffix $z$:

$$
\text{Prefix analysis} \quad \begin{cases} x : p_0 \ \vec{\Gamma}_\ell \ (\beta \ p_\ell \ \beta') \end{cases}
$$

$$
\text{Pumpable analysis} \quad \begin{cases} y_1 : \Phi_x \Rightarrow \Phi_{y1} \quad \text{where } \Phi_x = \mathsf{Set}(\beta \ p_\ell) \\ a : \Phi_{y1} \Rightarrow \Phi_{y1a} \\ y_2 : \Phi_{y1a} \Rightarrow \Phi_{y2} \ \text{where } \Phi_{y2} \subseteq \Phi_x \end{cases}
$$

$$
\text{Suffix analysis} \quad \begin{cases} z : \Phi_{y2} \Rightarrow \Phi_{\text{fail}} \ \text{where } \Phi_{\text{fail}} \cap \mathbf{Acc} = \emptyset \end{cases}
$$

## 3.2 Prefix analysis

The analysis needs to find a string that causes the matcher to reach $p_\ell$. However, due to the nondeterminism of the underlying NFA, it is not enough to check reachability. The same string $x$ could also lead to some other states before $p_\ell$ is reached by the matcher. If one of these states could lead to acceptance, the

$p_0$

$x$    $x$    $x$

$p_1$    $p_\ell$    $p_{\text{Right}}$

$y$    $y$    $y$

$p_2$    $p_\ell$    $p_\ell$

$y$    $y$   $y$    $y$   $y$

$p_3$    $p_\ell$   $p_\ell$    $p_\ell$   $p_\ell$

$z$    $z$   $z$    $z$   $z$

fail    fail   fail    fail   fail
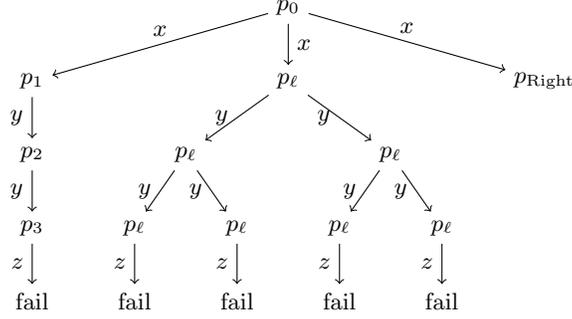
Fig. 2: Branching search tree with left context for $x\,y\,y\,z$

matcher will terminate successfully, and $p_\ell$ will never be reached. In this case, there is no vulnerability, regardless of any exponential blowup in the subtree under $p_\ell$. See Figure 2.

**Definition 7.** *The operator $\ggg$ removes all but the leftmost occurrences in sequences according to the following rules:*

$$[\,] \ggg [\,]$$
$$p\,\beta \ggg p\,\beta' \qquad \text{if } (\nexists \beta_1, \beta_2 \,.\, \beta = \beta_1\, p\, \beta_2) \wedge \beta \ggg \beta'$$
$$p\,\beta \ggg \beta' \qquad \text{if } (\exists \beta_1, \beta_2 \,.\, \beta = \beta_1\, p\, \beta_2) \wedge p\,\beta_1\,\beta_2 \ggg \beta'$$

Note that $\ggg$ is applied on shorter sequences on the R.H.S, ensuring termination. Moreover, in each reduced sequence each $p$ can appear at most once, so there are only finitely many sequences that can be reached in the REDoS prefix analysis (below).

**Definition 8.** *Let $p_\ell$ be the NFA state we are currently analyzing. The transition relation for ordered multistates is defined as follows:*

$$\frac{\beta_1 = (p_1 \ldots p_n) \qquad a : p_i \mapsto \theta_i \qquad (\theta_1 \ldots \theta_n) \ggg \beta_2}{a : \beta_1 \restriction_\ell \beta_2}$$

*The relation is extended to strings:*

$$\frac{w : \beta_1 \restriction_\ell \beta_2 \qquad a : \beta_2 \restriction_\ell \beta_3}{(w\,a) : \beta_1 \restriction_\ell \beta_3} \qquad \overline{\varepsilon : \beta \restriction_\ell \beta}$$

The REDoS prefix analysis computes all ordered multistates $\beta$ reachable from $p_0$, together with a realizer $w$, using the following rules:

$$\frac{}{(\varepsilon, p_0) \in \mathcal{R}} \qquad \frac{(w, \beta_1) \in \mathcal{R} \quad a : \beta_1 \restriction_\ell \beta_2 \quad \nexists w'.(w', \beta_2) \in \mathcal{R}}{(w\,a, \beta_2) \in \mathcal{R}}$$

In the implementation, we keep a set $\mathcal{R}$. It is initialized to $(\varepsilon, p_0)$. We then repeatedly check if there is a $(w, \beta_1)$ in the set such that for some $a$ there is a transition $a : \beta_1 \vdash_\ell \beta_2$. If there is, we add $(w\,a, \beta_2)$ to $\mathcal{R}$ and repeat the process. We terminate when no new $\beta_2$ has been found in the last iteration. Finally, the analysis isolates $(w, \beta)$ pairs of the form $(x, \beta\ p_\ell\ \beta')$ and takes $\Phi_x$ as $\mathsf{Set}(\beta\ p_\ell)$ for each such pair.

### 3.3 Pumping analysis

**Definition 9.** *A branch point is a tuple*

$$(p_N, a, \{p_{N1}, p_{N2}\})$$

*such that $p_{N1} \neq p_{N2}$, $a : p_N \twoheadrightarrow p_{N1}$ and $a : p_N \twoheadrightarrow p_{N2}$.*

For example, if $p_N$ has three successor nodes $p_1$, $p_2$ and $p_3$ for the same input symbol $a$, there are three different branch points:

$$(p_N, a, \{p_1, p_2\})$$
$$(p_N, a, \{p_1, p_3\})$$
$$(p_N, a, \{p_2, p_3\})$$

There can be only finitely many non-deterministic nodes in the given NFA. For each of them, we need to solve a reachability problem.

The pumping analysis can be visualized with the diagram in Figure 3. The analysis aims to find two different paths leading from $p_\ell$ to itself. Such paths must at some point include a nondeterministic node $p_N$ that has at least two transitions to different nodes $p_{N1}$ and $p_{N2}$ for the same symbol $a$. For such a node to lie on a path from $p_\ell$ to itself, there must be some path labeled $y_1$ leading from $p_\ell$ to $p_N$, and moreover there must be paths from the two child nodes $p_{N1}$ and $p_{N2}$ leading back to $p_\ell$, such that both these paths have the label $y_2$. The left side of Figure 3 depicts this situation.

So far we have only considered what states *may* be reached. Due to the nondeterminism of the transition relation $a : p \twoheadrightarrow q$, there may be other states that can be reached for the same strings $y_1$ and $y_2$. Therefore, we also need to perform a *must* analysis that keeps track of all states reachable via the strings we construct. This analysis uses the transition relation $\Rightarrow$ of the power DFA between sets of NFA states. In Figure 3, it is shown on the right-hand side.

Intuitively, we run the two transition relations in parallel on the same input string. More formally, this involves constructing a product of two relations. Before we reach the branching point, we run the relations $\twoheadrightarrow$ and $\Rightarrow$ in parallel. After the nondeterministic node $p_N$ has produced two different successors, we need to run two copies of $\twoheadrightarrow$ in parallel with $\Rightarrow$. One may visualize this situation by reading the diagram in Figure 3 horizontally: above the splitting at $p_N$, there are two arrows in parallel for $y_1$, whereas below that node, there are three arrows in parallel for $a$ and $y_2$.

The twofold transition relation $\twoheadrightarrow_2$ for running $\twoheadrightarrow$ in parallel with $\Rightarrow$ is given by the rules in Figure 4. Analogously, the threefold product transition relation $\twoheadrightarrow_3$ for running two copies of $\twoheadrightarrow$ in parallel with $\Rightarrow$ is given by the rules in Figure 5.

In summary, the pumping analysis consists of two phases:

1. Given $p_\ell$ and $\Phi_x$, the analysis searches for a realizer $y_1$ for reaching some nondeterministic node $p_N$:

$$y_1 : (p_\ell, \Phi_x) \twoheadrightarrow_2 (p_N, \Phi_{y1})$$

2. Given the successor nodes $p_{N1}$ and $p_{N2}$ of some $p_N$ node, the analysis searches for a realizer $y_2$ for reaching $p_\ell$:

$$y_2 : (p_{N1}, p_{N2}, \Phi_{y1a}) \twoheadrightarrow_3 (p_\ell, p_\ell, \Phi_{y2})$$

Moreover, the analysis checks that the constructed state $\Phi_{y2}$ satisfies the inclusion:

$$\Phi_{y2} \subseteq \Phi_x$$

If both phases of the analysis succeed, the string $y_1 \, a \, y_2$ is returned as the pumpable string, together with the state $\Phi_{y2}$.
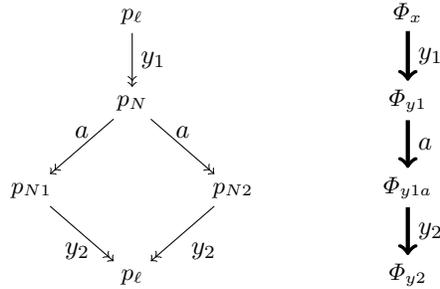


Fig. 3: Pumping analysis construction of $y_1 \, a \, y_2$: "may" on the left using $\twoheadrightarrow$, and "must" on the right using $\Rightarrow$

*Example 1.* The following diagram shows an NFA corresponding to the regular expression $(a|b|ab)^*$:
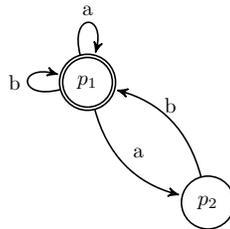
$$\frac{w : (p_1, \varPhi_1) \twoheadrightarrow_{\mathbf{2}} (p_2, \varPhi_2) \qquad \begin{array}{c} b : p_2 \twoheadrightarrow p_3 \\ b : \varPhi_2 \Rrightarrow \varPhi_3 \end{array}}{(w\,b) : (p_1, \varPhi_1) \twoheadrightarrow_{\mathbf{2}} (p_3, \varPhi_3)}$$

$$\overline{\varepsilon : (p, \varPhi) \twoheadrightarrow_{\mathbf{2}} (p, \varPhi)}$$

Fig. 4: The twofold product transition relation $\twoheadrightarrow_{\mathbf{2}}$

$$\frac{w : (p_1, p_1', \varPhi_1) \twoheadrightarrow_{\mathbf{3}} (p_2, p_2', \varPhi_2) \qquad \begin{array}{c} b : \;\; p_2 \twoheadrightarrow p_3 \\ b : \;\; p_2' \twoheadrightarrow p_3' \\ b : \varPhi_2 \Rrightarrow \varPhi_3 \end{array}}{(w\,b) : (p_1, p_1', \varPhi_1) \twoheadrightarrow_{\mathbf{3}} (p_3, p_3', \varPhi_3)}$$

$$\overline{\varepsilon : (p, p', \varPhi) \twoheadrightarrow_{\mathbf{3}} (p, p', \varPhi)}$$

Fig. 5: The threefold product transition relation $\twoheadrightarrow_{\mathbf{3}}$

Taking $p_\ell = p_1$ and $\varPhi_x = \{p_1\}$, the pumping analysis leads to the following derivation:

$$\begin{array}{cl} y_1 = \varepsilon & \varepsilon : (p_1, \{p_1\}) \twoheadrightarrow_{\mathbf{2}} (p_1, \{p_1\}) \\ (p_1, a, \{p_1, p_2\}) & a : \{p_1\} \Rrightarrow \{p_1, p_2\} \\ y_2 = b & b : (p_1, p_2, \{p_1, p_2\}) \twoheadrightarrow_{\mathbf{3}} (p_1, p_1, \{p_1, p_2\}) \end{array}$$

Here we have an *unstable* derivation since $\{p_1, p_2\} \nsubseteq \{p_1\}$ (i.e. $\varPhi_{y_2} \nsubseteq \varPhi_x$). If we were to take $\varPhi_x = \{p_1, p_2\}$ (i.e. $x = a$), the resulting derivation would be stable (for the same pumpable string $ab$). Stable derivations ensure that multiple pumpings of a pumpable string do not diverge $\varPhi_{y_2}$, which in turn ensures the correctness of the failure suffix. We treat this inclusion more formally in Lemma 7.

### 3.4 Suffix analysis

For each $\varPhi_{y2}$ constructed by the pumping analysis, the REDoS failure analysis computes all multistates $\varPhi_{\mathrm{fail}}$ such that there is a $z$ with:

$$z : \varPhi_{y2} \Rrightarrow \varPhi_{\mathrm{fail}} \qquad \wedge \qquad \varPhi_{\mathrm{fail}} \cap \mathbf{Acc} = \emptyset$$

Intuitively, $z$ *fails all* the states in $\varPhi_{y2}$ by taking them to $\varPhi_{\mathrm{fail}}$, which does not contain any accepting states.

## 4 Test cases for the REDoS analysis

In order to demonstrate the behavior of the analyser, here we present examples that exercise the most important aspects of its operation.

### 4.1 Non commutativity of alternation

This aspect of the analysis can be illustrated with the following two example expressions:

$$.^* \mid (a \mid b \mid ab)^* c$$

$$(a \mid b \mid ab)^* c \mid .^*$$

Even though the two expressions correspond to the same language, only the second expression yields a successful attack. In the first expression, all the multi-states starting from $\Phi_x$ ($\mathsf{Set}(\beta p_\ell)$) consist of a state corresponding to the expression $(.^*)$, which implies that this expression is capable of consuming any input string thrown at it without invoking the vulnerable Kleene expression. On the other hand, $\Phi_x$ calculated for the second expression lacks a state corresponding to $(.^*)$, leading to the following attack string configuration:

$$x = \varepsilon \quad y = ab \quad z = \varepsilon$$

### 4.2 Prefix construction

Prefix construction plays one of the most crucial roles in finding an attack string. In the following example, only a certain prefix leads to a successful attack string derivation:

$$c.^*\mid(c \mid d)(a \mid b \mid ab)^* e$$

Notice that a prefix $c$ would trigger the $(.^*)$ on the left due to the left-biased treatment of alternation in backtracking matchers. The prefix $d$ on the other hand forces the matcher out of this possibility. The difference between these two prefixes is captured in two different values of $(x, \Phi_x)$:

$$(c, \{p_1, p_2\}) \quad (d, \{p_2\})$$

Where

$$p_1 \models .^* \text{ and } p_2 \models (a \mid b \mid ab)^* e$$

Only the latter of these two leads to a successful attack string:

$$x = d \quad y = ab \quad z = \varepsilon$$

Prefix construction may also lead to loop unrolling when necessary. For an example, consider the following regex:

$$(a \mid b).^*\mid c^*(a \mid ab \mid b)^* d$$

Without the unrolling of the Kleene expression $c^*$, any pumpable string intended for the vulnerable Kleene expression will be consumed by the alternation on the left. The analyser captures this situation again as two different values of $(x, \Phi_x)$, one for $x = c$ and the other for either $x = a$ or $x = b$. Only the former value leads to a successful attack string:

$$x = c \quad y = ab \quad z = \varepsilon$$

The amount of loop unrolling is limited by the finite-ness of the $\Phi_x$ values. In the following example, the loop $c^*$ needs to be unrolled twice:

$$(c \mid a \mid b)(a \mid b).^* \mid c^*(a \mid b \mid ab)^* d$$

Here, unrolling $c^*$ 0 - 2 times leads to three distinct values of $\Phi_x$ due to the different matching states on the left alternation. Only one of those unrollings leads to a successful attack string:

$$x = cc \quad y = ab \quad z = \varepsilon$$

### 4.3 Pumpable construction

As is the case with prefixes, the existence of an attack string may depend on the construction of an appropriate pumpable string. For an example, consider the following regex:

$$(a \mid a \mid b \mid b)^*(a.^* \mid c)$$

Here the pumpable string $a$ does not yield an attack string since it also triggers the $(.^*)$ continuation. On the other hand, the pumpable string $b$ avoids this situation and leads to the following attack string configuration:

$$x = \varepsilon \quad y = b \quad z = \varepsilon$$

Similar to the prefix analysis, pumpable analysis utilises $(y, \Phi_y)$ values to select between pumpable strings.

In some cases, the pumpable construction overlaps with prefix construction. In the example below, an attack string may be composed in two different ways:

$$d.^* \mid ((c \mid d)(a \mid a))^* b$$

Here, choosing $ca$ as the pumpable string leads to a successful attack string derivation:

$$x = \varepsilon \quad y = ca \quad z = \varepsilon$$

However, it is also possible to form an attack string with the following configuration:

$$x = ca \quad y = da \quad z = \varepsilon$$

The important point here is that the attack string must begin with a $c$ instead of a $d$ in order to avoid the obvious match on the left. The analyser is capable of finding both the configurations that meet this requirement.

Pumpable construction may also lead to loop unrolling when necessary, as demonstrated by the following example:

$$a.^*|(c^*a(b\mid b))^*d$$

Without unrolling the inner loop $c^*$, the pumpable string $ab$ would trigger the alternation on the left. A successful attack string requires the unrolling of this inner loop, as in the following configuration:

$$x = \varepsilon \quad y = cab \quad z = \varepsilon$$

As with the previous example, the unrolling of the inner loop $c^*$ may be performed as part of the prefix construction, leading to the following alternate attack string configuration:

$$x = cab \quad y = ab \quad z = \varepsilon$$

The latter configuration may be considered more desirable in that it makes the the pumpable string shorter, leading to much smaller attack strings.

## 5   Soundness of the analysis

The backtracking machine performs a depth-first search of a search tree. Proofs about runs of the machine are thus complicated by the fact that the construction of the tree and its traversal are conflated. To make reasoning more compositional, we define a substructural calculus for constructing search trees. Machine runs correspond to paths from roots to leaves in these trees.

### 5.1   Search tree logic

**Definition 10 (Search tree logic).** *The search tree logic has judgements of the form*

$$w : \beta_1 \bigtriangleup \beta_2$$

*where $w$ is an input string, and both $\beta_1$ and $\beta_2$ are sequences of NFA states. The inference rules are given in Figure 6.*

Intuitively, the judgement

$$w : \beta_1 \bigtriangleup \beta_2$$

means that there is a horizontal slice of the search tree, such that the nodes at the top form the sequence $\beta_1$, the nodes at the bottom form the sequence $\beta_2$, and all paths have the same sequence of labels, forming $w$:
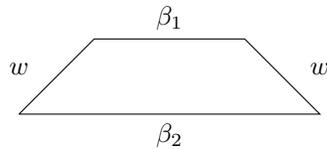
$$\frac{a \,:\, p \mapsto \beta}{a \,:\, p \,\triangle\, \beta} \,(\textsc{Trans1}) \qquad \frac{\not\exists \beta.a \,:\, p \mapsto \beta}{a \,:\, p \,\triangle\, \varepsilon} \,(\textsc{Trans2})$$

$$\frac{w_1 \,:\, \beta_1 \,\triangle\, \beta_2 \qquad w_2 \,:\, \beta_2 \,\triangle\, \beta_3}{(w_1\,w_2) \,:\, \beta_1 \,\triangle\, \beta_3} \,(\textsc{SeqComp})$$

$$\frac{}{\varepsilon \,:\, \beta \,\triangle\, \beta} \,(\varepsilon\textsc{Seq})$$

$$\frac{w \,:\, \beta_1 \,\triangle\, \beta_2 \qquad w \,:\, \beta_1' \,\triangle\, \beta_2'}{w \,:\, (\beta_1\,\beta_1') \,\triangle\, (\beta_2\,\beta_2')} \,(\textsc{ParComp})$$

$$\frac{}{w \,:\, \varepsilon \,\triangle\, \varepsilon} \,(\varepsilon\textsc{Par})$$

Fig. 6: Search tree logic

Each $w$ represents an NFA run $w \,:\, p_1 \twoheadrightarrow p_2$ for some $p_1$ that occurs in $\beta_1$ and some $p_2$ that occurs in $\beta_2$. The string $w$ labels the sides of the trapezoid, since that determines the compatible boundary for parallel composition. Again we may like to think of $w$ as a proof of reachability. Here the reachability is not in the NFA, but in the matcher based on it.

The trapezoid can be stacked on top of each other if they share a common $\beta$ at the boundary. They can be placed side-by-side if they have the same $w$ on the inside:
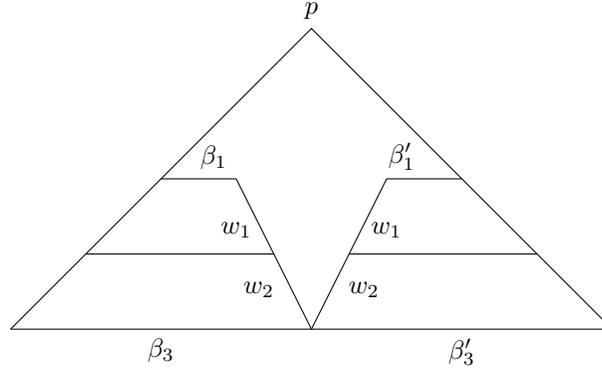


## 5.2 Pumpable implies exponential tree growth

We use the search tree logic to construct a tree by closely following the phases of our REDoS analysis. The exponential growth of the search tree in response to pumping is easiest to see when thinking of horizontal slices across the search tree for each pumping of $y$. The machine computes a diagonal cut across the search tree as it moves towards the left corner. The analysis constructs horizontal cuts with all states at the same depth. It is sufficient to show that the width of the search tree grows exponentially. The width is easier to formalize than the size.

We need a series of technical lemmas connecting different transition relations.

**Lemma 2.** *The following rule is admissible:*

$$\frac{w \,:\, \Phi_1 \Rrightarrow \Phi_2 \qquad w \,:\, \Phi_1' \Rrightarrow \Phi_2'}{w \,:\, (\Phi_1 \cup \Phi_1') \Rrightarrow (\Phi_2 \cup \Phi_2')}$$

**Lemma 3 ($\Rrightarrow \triangle$ simulation).** *If $w : \Phi_1 \Rrightarrow \Phi_2$, $w : \beta_1 \triangle \beta_2$ and $\Phi_1 = \mathsf{Set}(\beta_1)$, then $\Phi_2 = \mathsf{Set}(\beta_2)$.*

*Proof.* Suppose:

$$\beta_1 = (p_1 \ldots p_n) \qquad a : p_i \mapsto \theta_i$$

Then from the search tree logic we get $a : \beta_1 \triangle (\theta_1 \ldots \theta_n)$. Moreover, the definition of $\Rrightarrow$ implies $a : \{p_i\} \Rrightarrow \mathsf{Set}(\theta_i)$. Now, applying Lemma 2 gives:

$$a \,:\, \mathsf{Set}(\beta_1) \Rrightarrow \mathsf{Set}(\theta_1) \cup \ldots \cup \mathsf{Set}(\theta_n) = \mathsf{Set}(\theta_1 \ldots \theta_n)$$

Therefore, the result holds for strings of unit length. An induction on the length of $w$ completes the proof.

**Lemma 4 ($\vec{\Gamma}_\ell \triangle$ simulation).** *If $w : \beta_1 \,\vec{\Gamma}_\ell\, \beta_2$, $w : \beta_1' \triangle \beta_2'$ and $\beta_1' \ggg \beta_1$, then $\beta_2' \ggg \beta_2$.*

*Proof.* Suppose:

$$\beta_1' = (p_{11} \ldots p_{mk}) \qquad a : p_{ij} \mapsto \theta_{ij}$$

Where $p_{ij}$ corresponds to the $j$th occurrence of the state $p_i$. Equivalently:

$$p_{ij} = p_{i'j'} \iff i = i'$$

Given $\beta_1' \ggg \beta_1$, we deduce:

$$(p_{11} \ldots p_{mk}) \ggg (p_{11} \ldots p_{m1}) = \beta_1$$

Now, given $a : \beta_1 \,\vec{\Gamma}_\ell\, \beta_2$, the definition of $\vec{\Gamma}_\ell$ gives:

$$(\theta_{11} \ldots \theta_{m1}) \ggg \beta_2$$

On the other hand, $a : \beta_1' \triangle \beta_2'$ gives:

$$\beta_2' = (\theta_{11} \ldots \theta_{mk})$$

The definition of $\ggg$ can be generalized for multi-states, which leaves us with:

$$(\theta_{11} \ldots \theta_{mk}) \ggg (\theta_{11} \ldots \theta_{m1})$$

That is, we have shown:

$$\beta_2' = (\theta_{11} \ldots \theta_{mk}) \ggg (\theta_{11} \ldots \theta_{m1}) \ggg \beta_2$$

An induction on the length of $w$ completes the proof.

**Lemma 5 ($\twoheadrightarrow \triangle$ simulation).** *Given $w : p \twoheadrightarrow q$, there are sequences of states $\beta_1$ and $\beta_2$ such that $w : p \triangle \beta_1 \, q \, \beta_2$.*

*Proof.* The base case ($w = a$) holds from the definition of $\triangle$. For the inductive step, suppose $w : p \twoheadrightarrow q$ and $a : q \mapsto q'$. Then from the induction hypothesis we get $w : p \triangle \beta_1 \, q \, \beta_2$ for some $\beta_1, \beta_2$. Moreover, from the base case we have $a : q \triangle \beta_3 \, q' \, \beta_4$ for some $\beta_3, \beta_4$. Assuming $a : \beta_1 \triangle \beta_1'$ and $a : \beta_2 \triangle \beta_2'$ for some $\beta_1', \beta_2'$, the definition of $\triangle$ gives $wa : p \triangle \beta_1' \, \beta_3 \, q' \, \beta_4 \, \beta_{2'}$.

**Lemma 6 (Pumpable realizes non-linearity).** *Let $y$ be pumpable for some node $p_\ell$. Then there exist $\beta_1$, $\beta_2$, $\beta_3$ such that:*

$$y : p_\ell \triangle \beta_1 \, p_\ell \, \beta_2 \, p_\ell \, \beta_3$$

*Proof.* The pumpable analysis generates a string of the form:

$$y = y_1 \, a \, y_2$$

Where

$$y_1 : p_\ell \twoheadrightarrow p_N$$

$$a : p_N \mapsto (\beta \, p_{N1} \, \beta' \, p_{N2} \, \beta'')$$

$$y_2 : p_{N1} \twoheadrightarrow p_\ell \qquad y_2 : p_{N2} \twoheadrightarrow p_\ell$$

Now, Lemma 5 leads to the desired result.

**Lemma 7.** *Let $x, y$ be constructed from the prefix analysis and the pumpable analysis such that:*

$$x : p_0 \, \overset{\curvearrowright}{\vdash}_\ell (\beta \, p_\ell \, \beta')$$
$$y : \mathsf{Set}(\beta \, p_\ell) \Rightarrow \Phi_y \qquad \Phi_y \subseteq \mathsf{Set}(\beta \, p_\ell)$$

*Then the following holds for any natural number $n$:*

$$\Phi_{y^n} \subseteq \Phi_{y^{n-1}}$$

*Where $\Phi_{y^0} = \mathsf{Set}(\beta \, p_\ell)$ and $y^n : \Phi_{y^0} \Rightarrow \Phi_{y^n}$.*

*Proof.* By induction on $n$. Note that the base case ($n = 1$) holds by construction. For the inductive step, suppose $\exists \, q \in \Phi_{y^n}$, then from the definition of $\Phi_{y^n}$ we get $\exists \, p \in \Phi_{y^{n-1}} \, . \, y : p \twoheadrightarrow q$. Moreover, the induction hypothesis gives $\Phi_{y^{n-1}} \subseteq \Phi_{y^{n-2}}$. Therefore, we have $p \in \Phi_{y^{n-2}}$, which in turn implies $q \in \Phi_{y^{n-1}}$.

The importance of Lemma 7 is that it allows us to calculate a failure suffix $z$ independent of the number of pumping iterations; $\Phi_{y^n}$ can only shrink as $n$ increases.

**Lemma 8 (Exponential tree growth).** *Let $x, y, z$ be constructed from the analysis such that:*

$$x : p_0 \;\ulcorner_\ell (\beta \, p_\ell \, \beta')$$
$$y : \mathsf{Set}(\beta \, p_\ell) \Rightarrow \Phi_y \qquad \Phi_y \subseteq \mathsf{Set}(\beta \, p_\ell)$$
$$z : \Phi_y \Rightarrow \Phi_{\mathrm{fail}} \qquad \Phi_{\mathrm{fail}} \cap \mathbf{Acc} = \emptyset$$

*Then there exists $\beta_L, \beta_R$ such that:*

$$x : p_0 \,\triangle\, \beta_L \, p_\ell \, \beta_R \,\wedge\, \mathsf{Set}(\beta_L) = \mathsf{Set}(\beta) \tag{A}$$

$$y^n : \beta_L \, p_\ell \,\triangle\, \beta_n \Rightarrow |\beta_n| \geq 2^n \tag{B}$$

$$z : \mathsf{Set}(\beta_n) \Rightarrow \Phi' \Rightarrow \Phi' \cap \mathbf{Acc} = \emptyset \tag{C}$$

*Proof.* – **Statement (A)**: Suppose $x : p_0 \,\triangle\, \beta_x$. Then from Lemma 4 it follows that $\beta_x \ggg \beta \, p_\ell \, \beta'$. That is, $p_\ell$ must occur in $\beta_x$. If we dissect $\beta_x$ into $\beta_L \, p_\ell \, \beta_R$ such that $p_\ell \notin \mathsf{Set}(\beta_L)$, then from the definition of $\ggg$ it follows that $\mathsf{Set}(\beta_L) = \mathsf{Set}(\beta)$.
  – **Statement (B)**: Follows from Lemma 6. The number of copies of $p_\ell$ doubles at each pumping iteration.
  – **Statement (C)**: Suppose $y^n : \mathsf{Set}(\beta_L \, p_\ell) \Rightarrow \beta'_n$ . Since $\mathsf{Set}(\beta_L \, p_\ell) = \mathsf{Set}(\beta \, p_\ell)$ (statement A), Lemma 3 gives: $\mathsf{Set}(\beta_n) = \mathsf{Set}(\beta'_n)$. Now from Lemma 7 it follows that $\mathsf{Set}(\beta_n) \subseteq \Phi_y$. Since $z$ cannot lead to a successful match from any state in $\Phi_y$ (by construction), the same should be true for $\mathsf{Set}(\beta_n)$.

Lemma 8 may be visualized as in Figure 7. Note that the right hand slice of the tree (emanating from $\beta'$) is irrelevant, the depth-first strategy of a backtracking matcher forces it to explore the left hand slice first. Since none of the states at the bottom of the tree ($\beta'_n$) are accepting, it is forced to explore the (exponentially large, $|\beta_n| \geq 2^n$) entire slice (as proved in the following section).
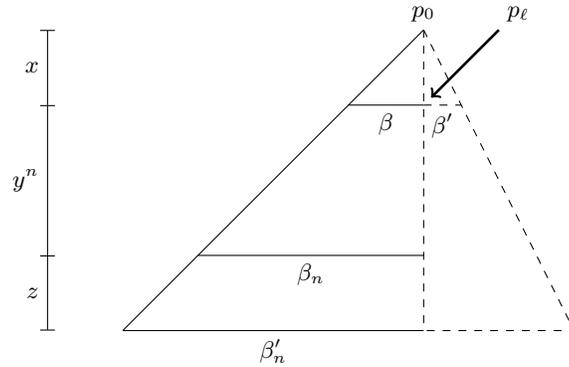


Fig. 7: Tree growth (Lemma 8)

### 5.3 From search tree to machine runs

Having proved that the attack strings lead to exponentially large search trees, in this section we show how backtracking matchers are forced to traverse all of it. We use the notation $w[i : j]$ to represent the substring of $w$ starting at index $i$ (inclusive) and ending at index $j$ (exclusive). That is,

$$w[i : i] = \varepsilon$$
$$w[i : j] = w[i]...w[j-1] \qquad (i < j)$$

**Lemma 9.** *Let $w$ be an input string of length $n$, $s$ a (constant) offset into $w$ $(0 \le s < n)$ and $p$ a state such that:*

$$w[s : i] : p \triangle \beta_i \quad s \le i \le n$$

$$\mathsf{Set}(\beta_n) \cap \mathbf{Acc} = \emptyset$$

*Then for any state $q$ appearing within some $\beta_i$, and for any $\sigma$, the following run exists:*

$$w \Vdash (q, i)\sigma \overset{*}{\rightsquigarrow} \sigma$$

*Proof.* By induction on $(n - i)$. For the base case $(i = n)$, we have the machine:

$$(q, n)\,\sigma$$

Since $q \in \beta_n$, this is not an accepting configuration. Therefore, we have:

$$w \Vdash (q, n)\,\sigma \rightsquigarrow \sigma$$

For the inductive step, suppose $i = k$ $(s \le k < n)$, then we have the machine:

$$w \Vdash (q, k)\,\sigma$$

If $q$ has no transitions on $w[k]$, the proof is trivial. Let us assume:

$$w[k] : q \mapsto q'_0 \dots q'_m$$

Then we have the transition:

$$w \Vdash (q, k)\,\sigma \rightsquigarrow (q'_0, k+1) \dots (q'_m, k+1)\,\sigma$$

Now the definition of $\triangle$ implies that $q'_0, \dots, q'_m$ are part of $\beta_{k+1}$. Therefore, we can apply the induction hypothesis to each of the newly spawned frames in succession, which leads to the desired result.

**Lemma 10.** *Let $w$ be an input string of length $n$, $s$ a (constant) offset into $w$ $(0 \le s < n)$ and $p$ a state such that:*

$$w[s : i] : p \triangle \beta_i \quad s \le i \le n$$

$$\mathsf{Set}(\beta_n) \cap \mathbf{Acc} = \emptyset$$

*Then for any state $q$ appearing within some $\beta_i$, the following run exists (for some $\sigma$):*

$$w \Vdash (p, s) \overset{*}{\rightsquigarrow} (q, i)\sigma$$

*Proof.* By induction on $(i - s)$. The base case $(i = s)$ holds trivially. For the inductive step, suppose $i = k$ $(s < k \le n)$ and that $\dot{q}$ appears in $\beta_k$. Then from the definition of $\triangle$, there must be some $q'$ appearing in $\beta_{k-1}$ such that:

$$\beta_{k-1} = \beta\, q'\, \beta' \qquad w[k-1] : q' \twoheadrightarrow q_0 \ldots \dot{q} \ldots q_m$$

Now from the induction hypothesis we get:

$$w \Vdash (p, s) \overset{*}{\leadsto} (q', k-1)\, \sigma$$

Therefore, we deduce the run:

$$w \Vdash (p, s) \overset{*}{\leadsto} (q', k-1)\sigma \leadsto (q_0, k) \ldots (\dot{q}, k) \ldots (q_m, k)\, \sigma$$

At this point, applying Lemma 9 to the newly spawned frames yields the required result.

Given a search tree with all failure nodes at the bottom, Lemma 9 shows that any intermediate frame reached during a simulation will eventually be rejected. Moreover, Lemma 10 shows that a simulation corresponding to such a search tree is forced to visit each and every node of the tree.

**Lemma 11 (Tree traversal).** *Suppose $w$ is an input string of length $n$ such that:*

$$w[0 : s] : p_0 \triangle \beta\, \overline{\beta}$$
$$w[s : n] : \beta \triangle \beta'$$
$$\mathsf{Set}(\beta') \cap \mathbf{Acc} = \emptyset$$

*Then for any state $q$ appearing in $\beta$, the following machine run exists (for some $\sigma$):*

$$w \Vdash (p_0, 0) \overset{*}{\leadsto} (q, s)\, \sigma$$

*Proof.* By induction on $s$. Note that the base case $(s = 0)$ follows from Lemma 10. For the inductive step, suppose $s > 0$. Here we focus on the lowest common ancestor of all the states in $\beta$, this situation is illustrated in the following figure:
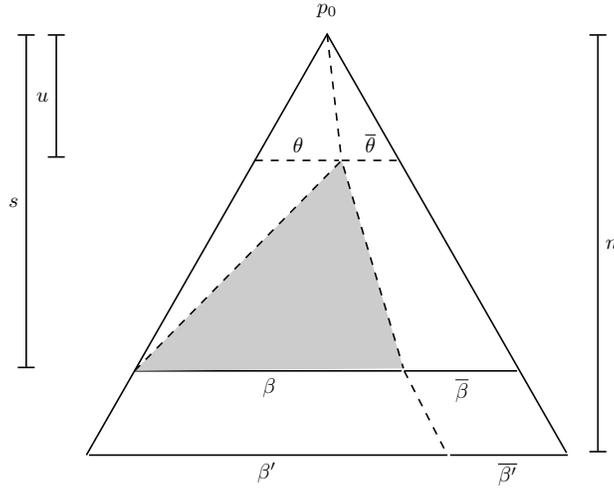
Let us assume that this state (lowest common ancestor of $\beta$) occurs at depth $u$ ($u > 0$, as otherwise we would have the base case again). Now from the diagram we deduce:

$$w[0 : u] : p_0 \,\triangle\, \theta \,\overline{\theta}$$
$$w[u : n] : \theta \,\triangle\, \beta'$$
$$\mathsf{Set}(\beta') \cap \mathbf{Acc} = \emptyset$$

Therefore, if $\dot{q}$ is the lowest common ancestor of $\beta$, from the induction hypothesis (since $u < s$) we get:

$$w \Vdash (p_0, 0) \overset{*}{\rightsquigarrow} (\dot{q}, u) \, \sigma$$

For some $\sigma$. Moreover, from Lemma 10 we deduce:

$$w \Vdash (\dot{q}, u) \overset{*}{\rightsquigarrow} (q, s) \, \sigma'$$

Where $q$ is some state in $\beta$ and $\sigma'$ is some failure continuation. Finally, we use Lemma 1 to compose these two runs into:

$$w \Vdash (p_0, 0) \overset{*}{\rightsquigarrow} (\dot{q}, u) \, \sigma \overset{*}{\rightsquigarrow} (q, s) \, \sigma' \, \sigma$$

In sum, we have shown that the pumped part of the search tree grows exponentially in the size of the input, and that the backtracking machine is forced to traverse all of it.

**Theorem 1 (Redos analyis soundness).** *Let the strings $x$, $y$ and $z$ be constructed by the REDoS analysis. Let $k$ be an integer. Then the backtracking machine takes at least $2^k$ steps on the input string $x\,y^k\,z$*

*Proof.* Follows from Lemma 8 and Lemma 11.

# 6 Completeness of the analysis

The analysis assumes that only a pumpable NFA can lead to an exponential runtime vulnerability. For completeness, we need to ensure that there are no other configurations that can cause such a vulnerability. Here we show that for any non-pumpable NFA, the width of any search tree is bounded from above by a polynomial. In places where an NFA is mentioned in a discussion below, a non-pumpable NFA is to be assumed (unless otherwise mentioned).

**Definition 11.** *For an ordered multi-state $\beta$ and a state $p$, we define the function $[\beta]_p$ as the number of ocurrences of $p$ within $\beta$. Moreover, the relations $\overset{p}{\sim}$ (p-simulate) and $\simeq$ (simulate) on ordered multi-states are incrementally defined as follows:*

$$\beta \overset{p}{\sim} \beta' \Leftrightarrow [\beta]_p = [\beta']_p$$

$$\beta \simeq \beta' \Leftrightarrow \forall\, p \in Q \,.\, \beta \overset{p}{\sim} \beta'$$

*It can be shown that both $\overset{p}{\sim}$ and $\simeq$ are reflexive, symmetric and transitive relations.*

**Lemma 12.** *The relation $\simeq$ can be shown to satisfy the following basic properties:*

$$\beta \simeq \beta' \Rightarrow \mathsf{Set}(\beta) = \mathsf{Set}(\beta') \wedge |\beta| = |\beta'|$$

$$\beta_1\beta_2 \simeq \beta_3\beta_4 \Leftrightarrow \forall\beta \,.\, \beta_1\beta\beta_2 \simeq \beta_3\beta\beta_4$$

$$\beta \simeq \beta_1\beta'\beta_2 \wedge \beta' \simeq \beta'' \Rightarrow \beta \simeq \beta_1\beta''\beta_2$$

$$\beta_1 \simeq \beta_1' \wedge \beta_2 \simeq \beta_2' \Rightarrow \beta_1\beta_2 \simeq \beta_1'\beta_2'$$

**Lemma 13.** *Let $w$ be an input string, $\beta_1$, $\beta_2$ be ordered multi-states such that:*

$$\beta_1 \simeq \beta_2 \quad w : \beta_1 \,\triangle\, \beta_1' \quad w : \beta_2 \,\triangle\, \beta_2'$$

*Then $\beta_1' \simeq \beta_2'$.*

*Proof.* Informally, $\beta_2$ is merely a re-ordering of $\beta_1$ (and vice versa). The trapezoid emanating from $\beta_1$ will be composed of individual search trees rooted at each constituent state of $\beta_1$. Therefore, the trapezoid emanating from $\beta_2$ will be a re-ordering of those search trees.

More formally, let $n = |\beta_1| = |\beta_2|$ (the latter equality holds since $\beta_1 \simeq \beta_2$). We perform an induction on $n$. The base case ($n = 1$) follows from the definition of $\triangle$ ($\beta_1 = \beta_2 = p$ for some $p \in Q$). For the inductive step, note that any state $q$ introduced to both $\beta_1$ and $\beta_2$ (to make them $n + 1$ in size) must be the same (in order to preserve $\beta_1 \simeq \beta_2$). Since the search tree rooted at $q$ is same for both $\beta_1$ and $\beta_2$ (regardless of where it appears within each of the multi-states), its contribution to $\beta_1'$ and $\beta_2'$ is the same.

**Definition 12.** *Given an NFA, a path $\gamma$ is a sequence of triples:*

$$(p_0, a_0, p_1)(p_1, a_1, p_2) \ldots (p_{n-1}, a_{n-1}, p_n)$$

*where for $0 \le i < n$, there is a transition $a_i : p_i \to p_{i+1}$ in the NFA. We write $\mathrm{dom}(\gamma)$ for the first node $p_0$ and $\mathrm{cod}(\gamma)$ for the last node $p_n$ in the path. The sequence of input symbols $a_0 \ldots a_n$ along the path is written as $\overline{\gamma}$ and called the label of the path. Moreover, the set of nodes $\{p_0, \ldots, p_n\}$ along the path $\gamma$ is written as $\mathsf{nodes}(\gamma)$.*

**Lemma 14.** *Given a tree judgement $w : p \bigtriangleup \beta$, for any state $q$ appearing in $\beta$, there exists a path $\gamma$ with $\mathrm{dom}(\gamma) = p$, $\mathrm{cod}(\gamma) = q$ and $\overline{\gamma} = w$.*

*Proof.* By induction on the length of $w$.

**Definition 13.** *We write $w : p \rightrightarrows q$ (p $\mathtt{two\text{-}paths}$ q) iff $\exists p_1, p_2, w_1, w_2$ such that:*

$$
\begin{aligned}
p_1 \ne p_2 \quad & w = w_1\, w_2 \\
w_1 : p \twoheadrightarrow p_1 \quad & w_1 : p \twoheadrightarrow p_2 \\
w_2 : p_1 \twoheadrightarrow q \quad & w_2 : p_2 \twoheadrightarrow q
\end{aligned}
$$

Definition 13 allows us to formulate pumpability in a different notation; if we have $w_1 : p \rightrightarrows q$ and $w_2 : q \twoheadrightarrow p$ (loop), then the state $p$ is pumpable on the input string $w_1 w_2$ (see Figure 3).

**Definition 14.** *Let $\gamma$ be a path. We define the sets $\mathcal{S}(\gamma)$ and $\mathcal{F}(\gamma)$ as follows:*

$$
\begin{aligned}
\mathcal{S}(\gamma) = \{p \mid {}& \exists\, \gamma_1, \gamma_2 \,.\, \gamma = \gamma_1 \gamma_2 \\
& \wedge\, \overline{\gamma}_2 : \mathrm{dom}(\gamma_2) \rightrightarrows \mathrm{cod}(\gamma_2) \wedge p = \mathrm{dom}(\gamma_2)\} \\
\mathcal{F}(\gamma) = {}& Q \setminus \mathcal{S}(\gamma)
\end{aligned}
$$

Essentially, $\mathcal{S}(\gamma)$ identifies the non-deterministic states along a path $\gamma$. There are at least two paths from a given state in $\mathcal{S}(\gamma)$ to $\mathrm{cod}(\gamma)$ bearing the same label ($\overline{\gamma}_2$ above, a suffix of $\overline{\gamma}$).

**Lemma 15.** *Suppose $\gamma$ is a path corresponding to a non-pumpable NFA such that $p = \mathrm{cod}(\gamma)$. Then the following holds for any $w$:*

$$w : p \bigtriangleup \beta \Rightarrow \mathsf{Set}(\beta) \subseteq \mathcal{F}(\gamma)$$

*Proof.* From the definitions we have:

$$\mathsf{Set}(\beta) \subseteq Q = \mathcal{S}(\gamma) \cup \mathcal{F}(\gamma)$$

Suppose $q \in \mathsf{Set}(\beta) \cap \mathcal{S}(\gamma)$. Then $q \in \mathcal{S}(\gamma)$ gives:

$$\exists\, w' \,.\, w' : q \rightrightarrows p$$

However, since $q \in \mathsf{Set}(\beta)$ we also have:

$$w : p \twoheadrightarrow q$$

Leading to the contradiction:

$$w'w : q \rightrightarrows p \twoheadrightarrow q$$

Therefore, it must be the case that $\mathsf{Set}(\beta) \cap \mathcal{S}(\gamma) = \emptyset$. This leads to the conclusion:

$$\mathsf{Set}(\beta) \subseteq \mathcal{F}(\gamma)$$
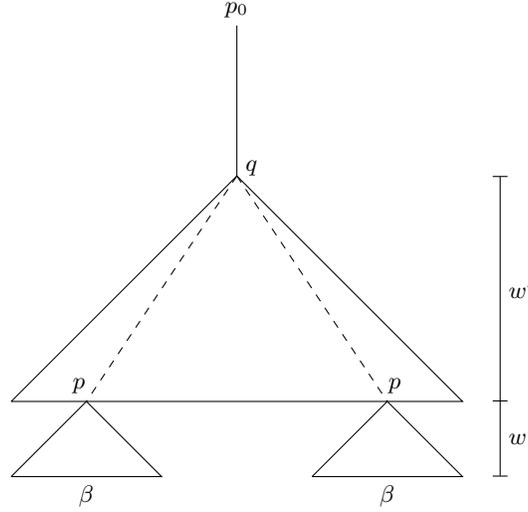
Fig. 8: Sibling restriction on $\mathcal{S}(\gamma)$

Lemma 15 is illustrated in Figure 8. Note that the fringes of the sibling trees rooted at the two $p$'s are identical ($\triangle$ logic is deterministic), making it impossible for either of them to contain a $q$ ($q$ would be pumpable otherwise). In other words, $q \in \mathcal{S}(\gamma)$ cannot appear again within search trees rooted at $\mathrm{cod}(\gamma)$. It is this restriction on non-determinism that leads us to the polynomial bound. However, flushing out this polynomial bound requires quite an elaborate analysis of the search tree structure, as we shall see next.

**Definition 15.** *We define the reduction $\triangleright$ on pairs of ordered multi-states according to the following rules:*

$$(q_1 \dots q_n, \beta_1 q \beta_2) \triangleright (q_1 \dots q_i q \dots q_n, \beta_1 \beta_2) \quad (\exists\, i\, .\, q = q_i)$$

$$(q_1 \dots q_n, \beta_1 q \beta_2 q \beta_3) \triangleright (q_1 \dots q_n q q, \beta_1 \beta_2 \beta_3) \quad (\nexists\, i\, .\, q = q_i)$$

*The reduction $\triangleright$ groups repeated states together. Given that each transition decreases the length of the second component, the reduction must terminate. We use the notation $\triangleright\triangleright$ to denote a maximal reduction:*

$$(\alpha_1, \beta_1) \triangleright\triangleright (\alpha_2, \beta_2) \Rightarrow \nexists (\alpha_3, \beta_3)\, .\, (\alpha_2, \beta_2) \triangleright (\alpha_3, \beta_3)$$

**Lemma 16.** *For a reduction $(\varepsilon, \beta) \triangleright \triangleright (\alpha, \sigma)$, the following basic properties can be shown to hold:*

$$\beta \simeq \alpha\sigma \tag{a}$$

$$\mathsf{Set}(\alpha) \cup \mathsf{Set}(\sigma) = \mathsf{Set}(\beta) \tag{b}$$

$$\forall\, p \in \mathsf{Set}(\alpha)\,.\,[\alpha]_p = [\beta]_p > 1 \tag{c}$$

$$|\sigma| = |\mathsf{Set}(\sigma)| \tag{d}$$

**Definition 16.** *We introduce an ordering variant of the search tree logic:*

$$w : (\beta, \alpha, \sigma)\bar{\triangle}(\beta', \alpha', \sigma')$$

*with the following inference rules:*

$$\frac{a : \beta_1\alpha_1 \triangle \beta_2 \quad a : \sigma_1 \triangle \beta_3 \quad (\varepsilon, \beta_3) \triangleright \triangleright (\alpha_2, \sigma_2)}{a : (\beta_1, \alpha_1, \sigma_1)\bar{\triangle}(\beta_2, \alpha_2, \sigma_2)}$$

$$\frac{w : (\beta_1, \alpha_1, \sigma_1)\bar{\triangle}(\beta_2, \alpha_2, \sigma_2) \qquad a : (\beta_2, \alpha_2, \sigma_2)\bar{\triangle}(\beta_3, \alpha_3, \sigma_3)}{wa : (\beta_1, \alpha_1, \sigma_1)\bar{\triangle}(\beta_3, \alpha_3, \sigma_3)}$$

The $\bar{\triangle}$ semantics recursively re-arranges the search tree into $\beta$, $\alpha$ and $\sigma$ components at each depth. A derivation using the $\bar{\triangle}$ semantics may be visualized as in Figure 9. Note that in this hypothetical derivation, we encounter repeated states at depth $w_1$, thus giving rise to the first non-empty $\alpha$ component ($\alpha_1$). From $w_1$ to $w_1w_2$, we have non-empty $\beta$ and $\sigma$ components. Again at depth $w_1w_2$ we can observe a non-empty $\alpha$ component, which is the result of the previous $\sigma$ component generating duplicates at this depth. The $\beta$ component can be thought of as the shadow/projection of all the previous $\alpha$ components.
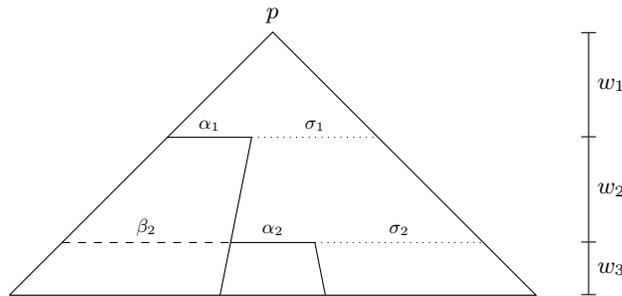


Fig. 9: An example $\bar{\triangle}$ derivation.

**Lemma 17.** *Let $p$ be a state and $w$ an input string such that:*

$$w : p \triangle \beta \qquad w : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta', \alpha, \sigma)$$

*Then $\beta'\alpha\sigma \simeq \beta$.*

*Proof.* By induction on the length of $w$. For the base case ($w = a$), suppose $a : p \triangle \beta$. Then from the definition of $\bar{\triangle}$ we get:

$$a : (\varepsilon, \varepsilon, p)\bar{\triangle}(\varepsilon, \alpha, \sigma)$$

Where $(\varepsilon, \beta) \triangleright\triangleright (\alpha, \sigma)$. Therefore, Lemma 16 (a) gives $\beta \simeq \alpha\sigma$. For the inductive step ($w = w'a$), suppose:

$$w' : p \triangle \beta_1 \tag{A.1}$$
$$w' : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta_1', \alpha_1, \sigma_1) \tag{A.2}$$

Then the induction hypothesis yields $\beta_1 \simeq \beta_1'\alpha_1\sigma_1$. Now let us assume:

$$a : \beta_1 \triangle \beta_2 \tag{B.1}$$
$$a : \beta_1'\alpha_1 \triangle \beta_2' \tag{B.2}$$
$$a : \sigma_1 \triangle \beta_{3'} \tag{B.3}$$
$$(\varepsilon, \beta_3') \triangleright\triangleright (\alpha_2, \sigma_2) \tag{B.4}$$

Assumptions (A.1), (B.2) - (B.4) and the definition of $\bar{\triangle}$ leads to:

$$w' : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta_2', \alpha_2, \sigma_2)$$

Moreover, assumptions (B.2), (B.3) implies

$$a : \beta_1'\alpha_1\sigma_1 \triangle \beta_{2'}\beta_{3'}$$

That is, we have:

$$\beta_1 \simeq \beta_1'\alpha_1\sigma_1 \tag{I.H}$$
$$a : \beta_1 \triangle \beta_2 \tag{B.1}$$
$$a : \beta_1'\alpha_1\sigma_1 \triangle \beta_{2'}\beta_{3'}$$

Applying Lemma 13 to these three relations yield $\beta_2 \simeq \beta_{2'}\beta_{3'}$. Furthermore, Lemma 16 (a) implies (with B.4) $\beta_{3'} \simeq \alpha_2\sigma_2$. Finally, Lemma 12 (c) gives $\beta_2 \simeq \beta_{2'}\alpha_2\sigma_2$ as required.

**Lemma 18.** *Let $\gamma$ be a path with $p = \text{cod}(\gamma)$ and $w$ an input string such that:*

$$w : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta, \alpha, \sigma)$$

*Then the following properties hold:*

$$\mathsf{Set}(\beta\alpha\sigma) \subseteq \mathcal{F}(\gamma) \tag{a}$$

$$|\sigma| \leq |\mathcal{F}(\gamma)| \tag{b}$$
$$|\alpha\sigma| \leq |\mathcal{F}(\gamma)| * o \tag{c}$$

*Proof.* For property (a), suppose $w : p \triangle \beta'$. From Lemma 15 we get $\mathsf{Set}(\beta') \subseteq \mathcal{F}(\gamma)$. Moreover, Lemma 17 gives $\beta\alpha\sigma \simeq \beta'$. Now, Lemma 12 (a) gives $\beta\alpha\sigma \subseteq \mathcal{F}(\gamma)$.

For property (b), note that it follows from property (a) that $\mathsf{Set}(\sigma) \subseteq \mathcal{F}(\gamma)$. From the definition of $\bar{\triangle}$ it follows that $\exists \beta' . (\varepsilon, \beta') \triangleright \triangleright (\alpha, \sigma)$. Therefore, from Lemma 16 (d) we get $|\sigma| = |\mathsf{Set}(\sigma)| \leq |\mathcal{F}(\gamma)|$.

For property (c), suppose $w = w'a$ (the result holds trivially for $w = \varepsilon$). Then from the definition of $\bar{\triangle}$ there exist $\sigma', \beta'$ such that:

$$w' : (\varepsilon, \varepsilon, p)\bar{\triangle}(\_, \_, \sigma') \tag{A.1}$$

$$a : \sigma' \triangle \beta' \tag{A.2}$$

$$(\varepsilon, \beta') \triangleright \triangleright (\alpha, \sigma) \tag{A.3}$$

From (A.2) and the structure of the NFA, we derive $|\beta'| \leq |\sigma'| * o$ (where $o$ is the fan-out of the NFA). Furthermore, (A.3) and Lemma 16 (a) implies $\alpha\sigma \simeq \beta'$. Therefore, Lemma 12 (a) and property (b) above leads to $|\alpha\sigma| = |\beta'| \leq |\sigma'| * o \leq |\mathcal{F}(\gamma)| * o$.

**Lemma 19.** *Let $w$ be an input string and $p$ a state. Let $k$ be a constant offset into $w$ and $i$ an index such that:*

$$0 < i \leq k \leq |w|$$

$$w[0 : i] : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta_i, \alpha_i, \sigma_i)$$

$$w[i : k] : \alpha_i \triangle \alpha_{(i,k)}$$

*Then $\beta_k = \alpha_{(1,k)} \ldots \alpha_{(k-1,k)}$*

*Proof.* By induction on $k$ (omitted).

With reference to Figure 9, Lemma 19 establishes the connection between the fringe of the overall triangle and those of individual trapezoidal slices (the concatenation of the bases of the trapezoids make up the base of the overall triangle).

**Lemma 20.** *Let $\gamma$ be a path with $p = \mathrm{cod}(\gamma)$ and $w$ an input string such that:*

$$w : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta, \alpha, \sigma)$$

*Then for a state $q$ appearing in $\alpha$, there exists a path $\gamma'$ from $p$ to $q$ such that $\mathcal{F}(\gamma\gamma') \subset \mathcal{F}(\gamma)$.*

*Proof.* Follows from Lemma 15 ($q$ is repeated within $\alpha$).

**Lemma 21.** *Suppose $\gamma$ is a path with $p = \mathrm{cod}(\gamma)$ and $w$ an input string of length $n$ such that:*

$$w : p \triangle \beta$$

*Then the following holds:*

$$|\beta| < k^k * o^k * n^k$$

*Where $k = |\mathcal{F}(\gamma)|$.*

*Proof.* By induction on $k$.

**Base case - 1**: Suppose $k = 0$. Then it follows from Lemma 15 that $|\beta| = 0$, which is within the bounds of our polynomial.

**Base case - 2**: Suppose $k = c$ (for some constant $c$) and:

$$\not\exists \gamma' \, . \, \gamma' = \gamma\gamma'' \wedge \mathcal{F}(\gamma') < c$$

This means the search tree rooted at $\text{cod}(\gamma)$ cannot contain duplicates at any depth, for if it does, we can always find an extended path $\gamma'$ for which $\mathcal{F}(\gamma')$ is less. This restriction immediately implies that the fringe of the search tree cannot grow beyond $c$, which is well within the bounds of our (over-estimating) polynomial ($c^c * o^c * n^c$).

**Inductive step**: Suppose:

$$w[0:i] : (\varepsilon, \varepsilon, p)\bar{\triangle}(\beta_i, \alpha_i, \sigma_i)$$

$$w[i:n] : \alpha_i \triangle \alpha_{(i,n)}$$

Where $0 < i \le n$. From Lemma 19 we deduce:

$$\beta_n \alpha_n \sigma_n = \alpha_{(1,n)} \ldots \alpha_{(n-1,n)} \alpha_n \sigma_n \tag{A}$$

It follows from Lemma 20 that we can apply the induction hypothesis to each path ending in some state within an $\alpha_i$. Therefore, we derive:

$$\forall i \; \exists v < k \; . \; \left|\alpha_{(i,n)}\right| < |\alpha_i| * v^v * o^v * |w[i:n]|^v$$

In terms of the illustration in Figure 9, this statement measures the bottom edges of the trapezoids. Now, taking into account that $v < k$ and $|w[i:k]| < n$, we arrive at:

$$\forall i \; . \; \left|\alpha_{(i,n)}\right| < |\alpha_i| * k^k * o^k * n^k$$

Moreover, it follows from Lemma 18 (c) that $|\alpha_i| \le k * o$. Therefore, we get:

$$\forall i \; . \; \left|\alpha_{(i,n)}\right| < k^{k+1} * o^{k+1} * n^k \tag{B}$$

Now, we combine (A) and (B) to obtain:

$$|\beta_n \alpha_n \sigma_n| < (n-1) * k^{k+1} * o^{k+1} * n^k + |\alpha_n \sigma_n|$$

Furthermore, it follows from Lemma 18 (c) that:

$$|\alpha_n \sigma_n| \le k * o < k^{k+1} * o^{k+1} * n^k$$

Therefore, we get:

$$|\beta_n \alpha_n \sigma_n| < k^{k+1} * o^{k+1} * n^{k+1}$$

Since we know $\beta \simeq \beta_n \alpha_n \sigma_n$ from Lemma 17, the inductive step holds.

**Theorem 2 (Redos analysis completeness).** *Given an NFA with an exponential runtime vulnerability, the REDoS analysis presented in Section 3 will produce an attack string which triggers this behaviour on a backtracking regular expression matcher.*

*Proof.* Lemma 21 implies that for a non-pumpable NFA, the search tree width is polynomially bounded. Since $w$ is finite, the entire search space in turn becomes polynomially bounded. This suggests that only a pumpable NFA can lead to an exponentially large search space. Finally, the analysis presented in Section 3 is exhaustive in that if a suitable attack string exists for a pumpable NFA, it will eventually be found.

## 7 Implementation

We implemented the analysis presented above in OCaml [30] (code-named *RXXR*). Apart from the code used for parsing regular expressions (and some other boiler-plate code), the main source modules have an almost one-to-one correspondence with the concepts discussed thus far. This relationship is illustrated in Table 10.

| Concept (Theory) | Implementation (OCaml Module) |
|---|---|
| NFA | `Nfa.mli/ml` |
| $\beta$ | `Beta.mli/ml` |
| $\Phi$ | `Phi.mli/ml` |
| $\twoheadrightarrow_2$ | `Product.mli/ml` |
| $\twoheadrightarrow_3$ | `Triple.mli/ml` |
| Prefix analysis | `XAnalyser.mli/ml` |
| Pumpable analysis ($y_1$) | `Y1Analyser.mli/ml` |
| Pumpable analysis ($ay_2$) | `Y2Analyser.mli/ml` |
| Suffix analysis | `ZAnalyser.mli/ml` |
| Overall analysis | `AnalyserMain.mli/ml` |

Fig. 10: Theory to source-code correspondence

Each module interface (`.mli` file) contains function definitions which directly correspond to various aspects of the analysis presented earlier. For an example, the NFA module provides the following function for querying ordered transitions:

```
val get_transitions : Nfa.t -> int ->
                      ((char * char) * int) list;;
```

The NFA states are represented as integers. Each symbol of the input alphabet is encoded as a pair of characters, allowing a uniform representation of character classes (`[a-z]`) as well as individual characters.

The NFA used in the implementation (`Nfa.mli/ml`) contains $\varepsilon$ transitions, which were not part of the NFA formalization presented earlier. The reason for this deviation is that having $\varepsilon$ transitions allows us to preserve the structure of the regular expression within the NFA representation, which in turn preserves the order of the transitions. The correctness of the implementation is unaffected as the two forms of NFA representation are isomorphic. Only a slight mental adjustment (from ordered NFAs to $\varepsilon$-NFAs) is required to correlate the theoretical formalizations to the OCaml code. For an example, Figure 11 presents the module interface for $\beta$. The function `advance()` is utilized inside the `XAnalyser.ml`

```
(* internal representation of beta *)
type t;;

module BetaSet : (Set.S with type elt = t);;

(* beta with just one state *)
val make : int -> t;;

(* returns the set of states contained within this beta *)
val elems : t -> IntSet.t;;

(* calculate all one-character reachable betas *)
val advance : (Nfa.t * Word.t * t) -> (Word.t * t) list;;

(* consume all epsilon transitions while recording pumpable
    kleene encounters *)
val evolve : (Nfa.t * Word.t * t) -> IntSet.t ->
                Flags.t * t * (int * t) list;;
```

Fig. 11: `Beta.mli`

module to perform the closure computation (i.e. compute all $\beta$s reachable from the root node), whereas `evolve()` is a utility function used to work around the $\varepsilon$ transitions. The modules (`Phi / Product / Triple`)`.mli` define similar interfaces for $\Phi, \twoheadrightarrow_2$ and $\twoheadrightarrow_3$ constructs introduced in the analysis.

The different phases of the analysis is implemented inside the corresponding analyser modules. As an example, Figure 12 presents the `Y2Analyser.mli` module responsible for carrying out the analysis after the branch point ($\twoheadrightarrow_3$ simulation). The internal representation of the analyser (`type t`) holds the state of the closure computation, which is initialized with an initial triple argument through the `init()` function. We defer the interested reader to module definition (`.ml`) files for further details on the implementation.

```
(* internal representation of the analyser *)
type t;;

(* initialize analyser instance for the specified triple and
    the kleene state *)
val init : (Nfa.t * Word.t * Triple.t) -> int -> t;;

(* calculate the next (y2, phi) *)
val next : t -> (Word.t * Phi.t) option;;

(* read analyser flags *)
val flags : t -> Flags.t;;
```

Fig. 12: `Y2Analyser.mli`

### 7.1 Evaluation data

The analysis was tested on two corpora of regexes. The first of these was extracted from an online regex library called *RegExLib* [23], which is a community-maintained regex archive; programmers from various disciplines submit their solutions to various pattern matching tasks, so that other developers can reuse these expressions for their own pattern matching needs. The second corpus was extracted from the popular intrusion detection and prevention system *Snort* [27], which contains regex-based pattern matching rules for inspecting IP packets across network boundaries. The contrasting purposes of these two corpora (one used for casual pattern matching tasks and the other used in a security critical application) allow us to get a better view of the seriousness of exponential vulnerabilities in practical regular expressions.

The regex archive for RegExLib was only available through the corresponding website [23]. Therefore, as the first step the expressions had to be scraped from their web source and adapted so that they can be fed into our tool. These adaptations include removing unnecessary white-space, comments and spurious line breaks. A detailed description of these adjustments as well as copies of both adjusted and un-adjusted data sets have been included with the resources linked from the RXXR distribution [30] (also including the Python script used for scraping). The regexes for Snort, on the other hand, are embedded within plain text files that define the Snort rule set. A Python script (also linked from the RXXR webpage) allowed the extraction of these regexes, and no further processing was necessary.

### 7.2 Results

The results of running the analysis on these two corpora of regexes are presented in Table 13. The figures show that we can process around 75% of each

of the corpora with the current level of syntax support. Out of these analyzable amounts, it is notable that regular expressions from the RegExLib archive use the Kleene operator more frequently (about 50% of the analyzable expressions) than those from the Snort rule set (close to 30%). About 11.5% of the Kleene-based RegExLib expressions were found to have a pumpable Kleene expression as well as a suitable suffix, whereas for Snort this figure stands around 0.55%.

| | RegExLib | Snort |
|---|---|---|
| Total patterns | 2992 | 12499 |
| Parsable | 2290 | 9801 |
| Pumpable | 159 | 19 |
| Vulnerable | 131 | 15 |
| Interrupted | 4 | 0 |
| Pruned | 0 | 2 |
| Time | 61.51 (s) | 30.10 (s) |

Fig. 13: RXXR2 results - statistics

The tool makes every attempt to analyse a given pattern, even the ones which contain non-regular constructs like backreferences. An expression $(e_1|e_2)$ may be vulnerable due to a pumpable Kleene that occurs within $e_1$, whereas $e_2$ might contain a backreference. In these situations, the analyser attempts to derive an attack string which avoids the non-regular construct. If such a non-regular construct cannot be avoided, the analysis is terminated with the `interrupted` flag.

On certain rare occasions, search pruning is employed as an optimization. It is activated when there have been a number of unstable derivations (failing to meet $\Phi_{y2} \subseteq \Phi_x$) for a given prefix. For an example, consider the regular expression:

$$([\hat{}a]^*b)^*[\hat{}c]\{1000\}$$

Here the Kleene expression $([\hat{}a]^*b)^*$ is pumpable for any string which contains two copies of $b$ (e.g. $bb, bab, abb, cbb\ldots$). However, if the analysis were to pick a pumpable string that *does not contain the symbol c*, it will lead to an unstable derivation. Intuitively, the followup expression $[\hat{}c]\{1000\}$ (which has a large state space) will also consume the pumpable string and introduce a new state in $\Phi_{y2}$, breaking the inclusion $\Phi_{y2} \subseteq \Phi_x$. Pruning allows the analysis to attempt different variants of the pumpable string without getting stuck on a single search path where all of the pumpable strings lead to unstable (but unique) derivations (e.g. $bb, bab, baab, baaab, \ldots$). Needless to say, this is an ad-hoc optimization that can be further improved with more sophisticated heuristics. Given that pruning was only triggered in two instances for the entire data set above, we believe the current heuristic (a static bound on the number of unstable derivations) is

adequate. If a pruned search does not report a vulnerability, it should be re-run with a higher (or infinite) prune limit in order to obtain a conclusive result.

**Validation** The task of validating vulnerabilities is complicated by the fact that different regular expression implementations (Java, Python, .NET etc.) have different syntax flavours. RXXR itself is written to accept PCRE like patterns of the form `/<REGEX>/<FLAGS>` where `REGEX` contains the main expression and `FLAGS` are used to control various aspects of the matching process (e.g. whether to match multi-line input or not). Java, Python and .NET use separate library calls to configure such behavior. Moreover, they can also differ from one another in terms of the syntax allowed within the main expression. For an example, Java requires tricky escape sequences when working with meta-characters (e.g a literal backslash requires \\\\), whereas Python is more flexible with its support for raw (un-interpreted) input strings.

For these reasons we chose Python as our main validation platform (Python's support for raw strings makes the porting relatively simple). A sample of vulnerabilities were then manually validated on other platforms (Java, .NET and PCRE). Table 14 illustrates how Python responds to above vulnerabilities.

|                        | RegExLib | Snort |
|------------------------|---------:|------:|
| Total vulnerabilities  | 131      | 15    |
| Successfully validated | 115      | 14    |
| Python parsing bug     | 12       | 0     |
| Python not vulnerable  | 4        | 1     |

Fig. 14: Validation of vulnerabilities - Python

The Python scripts developed for this validation are also included with the RXXR distribution [30], along with instructions on how to reproduce the above results. We discovered that Python was not able to compile regular expressions of the form $([a-z]^*)^*$, which is a known Python defect [32]. Variants of this bug affected 12 of the RegExLib vulnerabilities which we could not validate on Python. The remaining few cases were down to trivial vulnerabilities that Python manages to work around. We observed that both Python and .NET are capable of avoiding vulnerabilities in expressions like $([a-c]|b)^*d$ or $(a|a)^*b$, where the redundancies are quite obvious. Interestingly however, Java does not seem to implement any such workarounds; even when matching the expression $(a|a)^*b$ against the input string $a^n(n \sim 50)$, the JVM (Java Virtual Machine) becomes non-responsive.

**Sample vulnerabilities** The vulnerabilities reported range from trivial programming errors to more complicated cases. For an example, the following

regular expression is meant to validate time values in 24-hour format (from RegExLib):

```
^(([01][0-9]|[012][0-3]):([0-5][0-9]))*$
```

Here the author has mistakenly used the Kleene operator instead of the `?` operator to suggest the presence or non-presence of the value. This pattern works perfectly for all intended inputs. However, our analysis reports that this expression is vulnerable with the pumpable string "`13:59`" and the suffix "`/`". This result gives the programmer a warning that the regular expression presents a DoS security risk if exposed to user-malleable input strings to match.

For a moderately complicated example, consider the following regular expression (again from RegExLib):

```
^([a-zA-z]:((\\([-*\.*\w+\s+\d+]+)|(\w+)\\)+)(\w+.zip)|(\w+.ZIP))$
```

This expression is meant to validate file paths to zip archives. Our tool identifies this expression as vulnerable and generates the prefix "`z:\ `", the pumpable string "`\zzz\`" and the empty string as the suffix. This is probably an unexpected input in the author's eye, and this is another way in which our tool can be useful in that it can point out potential mis-interpretations which may have materialized as vulnerabilities.

Out of the over 12,000 patterns examined, there were two cases that failed to terminate within any reasonable amount of time. Closer inspection reveals that a pumpable Kleene expression with a vast number of states is to blame. Consider the following example (from RegExLib):

```
^(([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.
  ([a-zA-Z]{2,5}){1,25})+
  ([;.](([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.
  ([a-zA-Z]{2,5}){1,25})+)*$
```

If we change the counted expressions of the form `e{1,25}` into `e{1,5}`, the analyser returns immediately. This shows that the analysis itself can take a long time on certain inputs. However, such cases are extremely rare.

## 7.3 Comparison to fuzzers

REDoS analysers commonly used in practice are based on a brute-force approach known as fuzzing, where the runtime of a pattern is tested against a set of strings. A leading example of this approach is the Microsoft's SDL Regex Fuzzer [22].

As is common with most brute-force approaches, the main problem with fuzzing is that it can take a considerable amount of time to detect a vulnerability. This is especially pronounced in the case of REDoS analysis as vulnerable patterns tend to take increasing amounts of time with each iteration of testing. This property alone disqualifies fuzzing based REDoS analysers from being integrated into code-analysis tools, as their operation would impose unacceptable delays. For an example, consider the following simple pattern:

```
^(a|b|ab)*c$
```

Even with a lenient fuzzer configuration (ASCII only, 100 fuzzing iterations), SDL fuzzer takes 5-10 minutes to report a vulnerability on this pattern. By comparison, our analyser can process tens of thousands of patterns in less time.

Fuzzers can also miss out on vulnerabilities. For an example, consider the following two patterns:

```
^.*|(a|b|ab)*c$
^(a|b|ab)*c|.*$
```

SDL Fuzzer reports both of these patterns as being safe. However, the non-commutative property of the alternation renders the second pattern vulnerable (as explained in Section 4). Another such example is:

```
^(a|b|c|ab|bc)*a.*$
```

For this pattern, only one of the pumpable strings (*bc*) can lead to an attack string, and it must not end in an *a*. Such relationships are difficult to be caught in a heuristics-based fuzzer.

Yet another problem with fuzzers is caused by the element of randomness present in their string generating algorithms. Since fuzzers are not based on any sound theory, some form of randomness is necessary in order to increase the chance of stumbling upon a valid attack string. However, this can make the fuzzer yield inconsistent results for the same pattern. Consider the following pattern for an example:

```
(a|b)*[^c].*|(c)*(a|b|ab)*d
```

The SDL fuzzer reports this pattern as being safe in most invocations, but in few cases it finds an attack string.

Finally, the ultimate purpose of using a static analyser is to detect potential vulnerabilities upfront and lead to the corresponding fixes. Our analyser pinpoints the exact pumpable Kleene expression and generates a string (pumpable string) which witnesses vulnerability, making the fixing of the error a straightforward task. This is notably in contrast to the fuzzer, which outputs a random string (mostly in hex format) that does not provide any insight into the source of the problem.

## 8   Related work

The starting point for the present paper was the regular expression analysis RXXR [19]. While that paper was aimed at a security audience, the present paper complements it by using a programming language approach inspired by type theory and logic.

Program analysis for security is by now a well established field [5]. REDoS is known in the literature as a special case of algorithmic complexity attacks [8,26].

Parsing Expression Grammars (PEGs) have been proposed as an alternative to regular expressions [10] that avoid their nondeterminism. In a series of tutorials [6,7], Cox has argued for Thompson's lockstep matcher [31] as a superior alternative to backtracking matchers. However, backtracking matchers vulnerable to REDoS are still widely deployed, including the matchers in the Java and .NET platforms as well as the PCRE matcher used in some intrusion detection systems. Hence the REDoS problem will remain with us for the foreseeable future.

Backtracking is a classic application of continuations, and regular expression matchers similar to the backtracking machine have been investigated in the functional programming literature [9,14,11]. Other recent work on regular expressions in the programming language community includes regular expression inclusion [15] and submatching [28].

Apart from some basic constructions like the power DFA covered in standard textbooks [17], we have not explicitly relied on automata theory. Instead, we regarded the matcher as an abstract machine that can be analyzed with tools from programming language research. Specifically, the techniques in this paper are inspired by substructural logics, such as Linear Logic [12,13] and Separation Logic [18,24]. Concerning the latter, it may be instructive to compare the sharing of $w$ or absence of sharing of $\beta$ in Figure 6 to the connective of Separation logic. In a conjunction, the heap $h$ is shared:

$$\frac{h \models P_1 \qquad h \models P_2}{h \models P_1 \wedge P_2}$$

By contrast, in a separating conjunction, the heap is split into disjoint parts that are not shared:

$$\frac{h_1 \models P_1 \qquad h_2 \models P_2 \qquad h_1 \cap h_2 = \emptyset}{h_1 \cup h_2 \models P_1 * P_2}$$

Tree-shaped data structures have been one of the leading examples of separation logic and variations of it, such as Context Logic [3]. However, a difference to the search trees we have used in this paper is that the whole search tree is not actually constructed as a data structure in memory. Rather, only a diagonal cut across it is maintained at any time in the backtracking machine. The whole tree does not exist in memory, but only in space *and* time, so to speak. In that regard the search trees are like parse trees, which the parser only needs to construct in principle by traversing them, and not necessarily as a data structure in memory complete with details of all nodes [2,1].

Even though the backtracking machine is sequential, parts of the analysis are reminiscent of transition systems in process algebras, particularly running two or more automata in parallel (Figures 4 and 5). Seen that way, the may and must part of the analysis are analogous to the two modalities $\langle a \rangle$ and $[a]$ in Henessy-Milner logic [16].

## 9  Directions for further research

At present, the analysis constructs attack strings when there is the possibility of exponential runtime. It should be possible to extend the analysis to compute a polynomial as an upper bound for the runtime when there is no REDoS vulnerability causing exponential runtime.

The efficiency of the analyser compares favorably with that of the Microsoft SDL Regex Fuzzer [22]. Given that we are computing sets of sets of states, the analysis may explore a large search space. One may take some comfort from the fact that type checking and inference for functional programming languages can have high complexity in the worst case [21,25] that may not manifest itself in practice. Nonetheless, we aim to revisit the design of the analysis and optimize it.

Pruning the search space may lead to improvements in efficiency. An intriguing possibility is to implement the analysis on many-core graphics hardware (GPUs). Using the right data structure representation for transitions, GPUs can efficiently explore nondeterministic transitions in parallel, as demonstrated in the iNFAnt regular expression matcher [4].

The search tree logic (Figure 6) may have independent interest and possible connections to other substructural logics such as Linear Logic [12,13], Separation Logic [18,24], Lambek's syntactic calculus [20], or substructural calculi for parse trees [29]. Search trees are dual to parse trees in the sense that the nodes represent a disjunction rather than a conjunction.

## References

1. Aho, A.V., Lam, M., Sethi, R., Ullman, J.D.: Compilers - Principles, Techniques and Tools. Addison Wesley, second edn. (2007)
2. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java. Cambridge University Press (1997)
3. Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. In: POPL. pp. 271–282 (2005)
4. Cascarano, N., Rolando, P., Risso, F., Sisto, R.: iNFAnt: NFA Pattern Matching on GPGPU Devices. SIGCOMM Comput. Commun. Rev. 40(5), 20–26 (Oct 2010), `http://doi.acm.org/10.1145/1880153.1880157`
5. Chess, B., McGraw, G.: Static analysis for security. Security & Privacy, IEEE 2(6), 76–79 (2004)
6. Cox, R.: Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, Php, Python, Ruby, ...). Available at `http://swtch.com/~rsc/regexp/regexp1.html` (January 2007)
7. Cox, R.: Regular expression matching: the virtual machine approach. Available at `http://swtch.com/~rsc/regexp/regexp2.html` (December 2009)
8. Crosby, S.A., Wallach, D.S.: Denial of Service via Algorithmic Complexity Attacks. In: Proceedings of the 12th USENIX Security Symposium. Washington, DC (Aug 2003)
9. Danvy, O., Nielsen, L.R.: Defunctionalization at Work. In: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative

programming. pp. 162–174. PPDP '01, ACM, New York, NY, USA (2001), `http://doi.acm.org/10.1145/773184.773202`

10. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Principles of Programming Languages (POPL 2004). pp. 111–122. ACM (2004)
11. Frisch, A., Cardelli, L.: Greedy regular expression matching. In: ICALP. pp. 618–629 (2004)
12. Girard, J.Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
13. Girard, J.Y., Lafont, Y., Taylor, P.: Proof and Types. Cambridge University Press (1989)
14. Harper, R.: Proof-Directed Debugging. J. Funct. Program. 9(4), 463–469 (Jul 1999), `http://dx.doi.org/10.1017/S0956796899003378`
15. Henglein, F., Nielsen, L.: Regular expression containment: coinductive axiomatization and computational interpretation. In: Principles of Programming Languages (POPL 2011). pp. 385–398 (2011)
16. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: ICALP. pp. 299–309 (1980)
17. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
18. Ishtiaq, S.S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: Principles of Programming Languages (POPL). pp. 14–26. ACM (2001)
19. Kirrage, J., Rathnayake, A., Thielecke, H.: Static analysis for regular expression denial-of-service attacks. In: International Conference on Network and System Security (NSS 2013). pp. 135–148. No. LNCS 7873 in LNCS, Springer (2013)
20. Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly 65(3), 154–170 (1958)
21. Mairson, H.: Deciding ML typability is complete for deterministic exponential time. In: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 382–401. ACM (1989)
22. Microsoft: SDL Regex Fuzzer. Available at `http://www.microsoft.com/en-gb/download/details.aspx?id=20095` (2011)
23. RegExLib.com: Regular Expression Library. Available at `http://regexlib.com/` (2012)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74. IEEE (2002)
25. Seidl, H.: Haskell overloading is DEXPTIME-complete. Information Processing Letters 52(2), 57–60 (1994)
26. Smith, R., Estan, C., Jha, S.: Backtracking Algorithmic Complexity Attacks Against a NIDS. In: Proceedings of the 22nd Annual Computer Security Applications Conference. pp. 89–98. ACSAC '06, IEEE Computer Society, Washington, DC, USA (2006), `http://dx.doi.org/10.1109/ACSAC.2006.17`
27. Sourcefire: Snort (IDS/IPS). Available at `http://www.snort.org/` (2012)
28. Sulzmann, M., Lu, K.Z.M.: Regular expression sub-matching using partial derivatives. In: PPDP. pp. 79–90 (2012)
29. Thielecke, H.: On the semantics of parsing actions. Science of Computer Programming 84, 52–76 (2014)
30. Thielecke, H., Rathnayake, A.: RXXR2: Regular expression denial of service (ReDoS) static analysis. Available at `http://www.cs.bham.ac.uk/~hxt/research/rxxr2/index.shtml` (2014)
31. Thompson, K.: Programming Techniques: Regular Expression Search Algorithm. Communications of the ACM 11(6), 419–422 (June 1968)

32. Tracker, P.B.: Issue 2537: re.compile(...) should not fail. Available at `http://bugs.python.org/issue2537` (2008)