Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA

Nicolaas Weideman[1,4], Brink van der Merwe[1], Martin Berglund[2], and Bruce Watson[3]

[1] Department of Computer Science, Stellenbosch University, South Africa
[2] Department of Computer Science, Umeå University, Sweden
[3] FASTAR Research, Information Science, Stellenbosch University, South Africa
[4] Center for AI Research, CSIR, Stellenbosch University, South Africa

Abstract. We apply results from ambiguity of non-deterministic finite automata to the problem of determining the asymptotic worst-case matching time, as a function of the length of the input strings, when attempting to match input strings with a given regular expression, where the matcher being used is a backtracking regular expression matcher.

Keywords: regular expression, backtracking matcher, ambiguity

1 Introduction

Catastrophic backtracking is a phenomenon that causes extended matching time, when attempting to match certain input strings with so-called vulnerable regular expressions, when using backtracking regular expression matchers found in programming languages such as Java, Perl and Python. It can be used to launch regular expression denial of service (ReDoS) attacks, and there are numerous online accounts (some listed at [2]) of the occurrence of catastrophic backtracking. Catastrophic backtracking often occurs (although not necessarily or exclusively) when matching with a regular expression R, containing a subexpression S^* (or S^+), where S could match some non-empty input string w in multiple ways. Thus an input string containing w^k (i.e. k copies of w) as substring, may potentially be matched (or attempted to be matched) in exponentially (in k) many ways by R, in cases where the matcher tries most of the possible ways (one after the other, i.e. not using the subset construction) in which the substring w^k can be matched, in an attempt to obtain an overall match. Even though some regular expression matcher implementations do not match input strings in a backtracking fashion [8], these alternative implementations typically do not support all extended regular expression functionality that programmers have become accustomed to, such as back references.

Although catastrophic backtracking is typically regarded as being synonymous with exponential worst-case matching time, non-linear polynomial worst-case matching time might still be unsatisfactory from a performance or security

point of view. We regard non-linear matching time, vulnerable regular expressions and catastrophic backtracking to be all equivalent. We even point out cases with constant backtracking or matching time, where the constant is so large that the regular expression should be regarded as being vulnerable (from a practical point of view). Non-linear worst-case matching time often occurs when matching with a regular expression R, where R contains one or more occurrences of subexpressions of the form S^*U^* (or more generally S^*TU^*), where S and U (S, T and U) matches some common non-empty input string, say w. Similar to the exponential case, an input string containing w^k as substring, may now be matched (or attempted to be matched) in at least linearly (in k) many ways. The degree of the polynomial describing the worst-case number of ways in which an input string (of a given length) can be matched, depends on the number of occurrences of subexpressions of the form S^*U^* or S^*TU^*, where it is possible to move from one of these subexpressions (in the corresponding non-deterministic finite automaton) to the next while reading some input string. For example, a regular expression containing a subexpression $S_1^*U_1^*S_2^*U_2^*$, where S_i and U_i, for $i = 1, 2$, matches some non-empty input string w_i, could potentially attempt to match an input string containing $w_1^k w_2^k$, as substring, in quadratic (in k) different ways, leading to cubic matching time. To better understand the relationship between polynomial backtracking and matching time, consider a regular expression of the form S^*U^*, where S and U match some common non-empty input string w, but not any string of the form $w^k x$, for some suffix x. The matcher will first try to obtain an overall match by matching all of w^k with S^*, then backtrack and match only w^{k-1} with S^*, and continue this process of attempting to obtain an overall match by matching fewer and fewer of the repetitions of w with S^*, until S^* matches only the empty string. Thus since U^* first matches the empty string, then the last repetition of w, then the last two repetitions of w, etc., until it matches all of w^k, the matching time is quadratic in k, and thus in the length of $w^k x$.

A necessary condition to have exponential worst-case matching time is that the non-deterministic finite automaton (NFA), corresponding to the regular expression under consideration, contains a state with at least two loops that can be followed while processing the same substring (in a given input string). This condition is necessary and sufficient (under the additional assumptions that the NFA is trim and does not contain ε-loops) for an NFA to be exponentially ambiguous, i.e. to have input strings that can be matched in exponentially many ways in terms of their length ([5]). A necessary condition to have non-linear polynomial worst-case matching time, is that the corresponding NFA contains one or more pairs of states, such that for each pair of states p, q, there exists a string $w_{p,q}$ and loops on p and q and a path from p to q, all that can be followed while reading $w_{p,q}$. Let d be the length of the longest sequence of pairs of states, with the above properties, obtained by ordering the pairs of states such that there exists a path from the second state in a pair of states, to the first state in the next pair of states. Then $(d + 1)$ is the maximum degree of the polynomial describing the worst-case matching time. Again, if the NFA under consideration

is not exponentially ambiguous, these conditions are necessary and sufficient for the NFA to have polynomial ambiguity of degree d [5].

In the exponential matching time case, we refer to the part of the input string that can be matched in multiple ways while following these loops, as a *pump*, a string prefixed to the pump to ensure that the NFA reaches one of these states with two or more loops, as the *prefix*, and the string that is appended after the pump to ensure that the matcher attempts to match the pump in all possible ways, as the *suffix*. Thus exploit strings will be of the form $pw^k s$, with p the prefix, w the pump and s the suffix. In the non-linear polynomial matching time case, we have a pump for each subexpression of the form S^*U^* (or S^*TU^*). The strings required to move from the second state of a pair of states to the first state of the next pair of states, are referred to as the *pump separators*. Exploit strings will thus be of the form $s_0 w_1^k s_1 w_2^k \dots s_{n-1} w_n^k s_n$, $k \geq 0$, where the w_i's are the pumps, s_0 the prefix, $s_1, \dots, s_{n-1}$ the separators, and s_n the suffix. Again, the exploit strings correspond to strings exhibiting ambiguity, of a given form, in the underlying NFA (although strictly speaking, an additional sink accept state, having ε incoming transitions from all states, should be added to the underlying NFA, to make the correspondence between worst-case matching time and ambiguity precise).

As an example of a vulnerable regular expression, consider the following expression used to validate email addresses [4]:

```
R:=^([a-zA-Z0-9_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$
```

In the case of `R`, the subexpression $S :=$ `([a-zA-Z0-9]{2,4})+` can match the input string `aaaa` in two ways, either by matching `aaaa` by using the `+` in `S` once, or by using `+` twice by matching each time only `aa`. Note that this vulnerable regular expression is of a slightly different form than those described earlier, since in this case it is S and not `([a-zA-Z0-9]{2,4})`, that matches some input string in more than one way. We construct an input string capable of exploiting this vulnerability as follows. First, we construct a prefix capable of taking the matcher to the vulnerable subexpression, for example `a@a.` should suffice. Next, we add multiple repetitions of the pump `aaaa`. Finally, we force the matcher to reject our specifically crafted string. For this we append, for example, a '`$`' to the end of the input string. Strings of the form `a@a.`$(\texttt{aaaa})^k$`$` can thus be used as exploit strings.

For backtracking regular expression matchers, the different paths which can be traversed to possibly obtain an overall match, are prioritized, and also explored in this prioritized order, one after the other. Also, the matcher will not continue exploring alternative ways of matching the input string, after a match has been found. Consequently, regular expressions that seem very similar, even that match precisely the same language, may have completely different matching time behavior. Consider for example regular expressions of the form $R_1 := S \mid .^*$ and $R_2 := .^* \mid S$, where S has exponential worst-case matching time and '.' is the wild card symbol that matches any single input symbol. These regular expressions are equivalent in terms of languages matched, but not in terms of matching time, due to the fact that in R_1, matching will first be attempted with S, while in

R_2, the subexpression $.^*$ will be used first and S will be ignored. A slightly more complicated example is obtained by changing $.^*$ to $.\{m,\}$, i.e. an expression that matches strings of length m or more, with m a positive integer constant, in the regular expressions R_1 and R_2. In $R_2' := .\{m,\} \mid S$, the subexpression S with exponential worst-case matching time will now be reachable (in the corresponding non-deterministic finite automaton), but only for input strings of length shorter than m, leading to a regular expression with linear matching time. A non-trivial example of a similar type is `(\&d[0-9]{2}=.*?)+`, discussed in Section 4.

This paper extends results from [6]. We also consider how to determine the degree of the polynomial describing the worst-case matching time of a regular expression (if worst-case matching time is polynomial), which is listed as future work in [10]. The outline of the paper is as follows. In the next section we give the required definitions. After that, we provide our main results on deciding worst-case matching time behavior of a given regular expression, when using a backtracking regular expression matcher. Finally, we discuss our experimental results and conclude with a discussion on future work.

2 Definitions

In this section we introduce the notation and definitions required for the remainder of the paper. We denote by Σ a non-empty finite alphabet, which is used as input alphabet for automata and also an alphabet over which regular expressions are defined. As usual, ε denotes the empty word, and Σ^ε is used for $\Sigma \cup \{\varepsilon\}$. Also, Σ^* is the Kleene closure applied to Σ, thus the set of finite words over Σ. For $\Sigma_1 \subseteq \Sigma$ and $w = a_1 \dots a_n \in \Sigma^*$, with $a_i \in \Sigma$, we let $\pi_{\Sigma_1}(w)$ be the word $b_1 \dots b_n \in \Sigma_1^*$, with $b_i = a_i$ if $a_i \in \Sigma_1$, and $b_i = \varepsilon$ otherwise. For a function $f : A \to B$, and $a \in A$ and $b \in B$, we have that $f_{a \mapsto b} : A \to B$ is the function such that $f_{a \mapsto b}(a) = b$ and $f_{a \mapsto b}(x) = f(x)$ for all $x \in A \setminus \{a\}$. Also, $b^A : A \to B$, with $b \in B$, denotes the constant function with $f(x) = b$ for all $x \in A$. We use $\mathbb{N}$ for the set natural numbers, excluding 0. We denote by $|Q|$ the cardinality of the set Q and $\mathcal{P}(Q)$ the power set of Q.

A regular expression over an alphabet Σ (where $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \mid E')$, $(E \cdot E')$, or (E^*), where E and E' are regular expressions. Some parentheses can be dropped with the rule that * (Kleene closure) takes precedence over $\cdot$ (concatenation), which takes precedence over $\mid$ (union). Further, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language of a regular expression E, denoted $\mathcal{L}(E)$, is obtained by evaluating E as usual. When we say that E matches a string w, we mean that $w \in \mathcal{L}(E)$, as opposed to $vwv' \in \mathcal{L}(E)$, for $v, v' \in \Sigma^*$. Some of our examples of expressions will use operators other than just union, concatenation and Kleene star, but we will refer to all regular expressions, including the extended expressions, simply as *regexes* in the remainder of the paper.

A *tree* with labels in a set Σ is a function $t \colon V \to \Sigma$, where $V \subseteq \mathbb{N}^*$ is a non-empty, finite set of vertices (or nodes) which are such that (i) V is prefix-closed,

i.e., for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, $vi \in V$ implies $v \in V$, and (ii) V is closed to the left, i.e., for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, $v(i+1) \in V$ implies $vi \in V$. The vertex ε is the root of the tree and vertex vi is the ith child of v. We let $|t| = |V|$ be the size of t. We denote by t/v the tree t' with vertex set $V' = \{w \in \mathbb{N}^* \mid vw \in V\}$, where $t'(w) = t(vw)$ for all $w \in V'$. Given trees $t_1, \ldots, t_n$ and a symbol α, we let $\alpha[t_1, \ldots, t_n]$ denote the tree t with $t(\varepsilon) = \alpha$ and $t/i = t_i$ for all $i \in \{1, \ldots, n\}$.

Next we define non-deterministic finite automata (and runs for them), followed by the prioritized finite automata from [6] and [7], which are used to model regex matching behaviors exhibited by typical software implementations. In the definition of an NFA below, the transition function δ is defined to allow for parallel transitions on the same symbol between a pair of states. By $\delta(p, \alpha, q) = i > 0$, we indicate that there are i transitions on α between p and q. It is assumed that the transitions (if any) between p and q on α are numbered from 1 to $\delta(p, \alpha, q)$. We indicate by $p \rightarrow^{\alpha(j)} q$ (or $p\alpha(j)q$) that the jth-transition on α between p and q is taken. In our investigation, all parallel edges will be on ε, and we simply use $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_n$, instead of $\varepsilon(1), \varepsilon(2), \ldots \varepsilon(n)$. Although parallel transitions do not influence the language accepted by an NFA, they do influence the number of accepting paths of a given input string, and thus play a role in our setting.

Definition 1. *A* non-deterministic finite automaton *(NFA) is a tuple* $A = (Q, \Sigma, q_0, \delta, F)$ *where: (i)* Q *is a finite set of* states; *(ii)* Σ *is the* input alphabet; *(iii)* $q_0 \in Q$ *is the* initial state; *(iv) the partial function* $\delta : Q \times \Sigma^\varepsilon \times Q \rightarrow \mathbb{N}$ *is the* transition function; and *(v)* $F \subseteq Q$ *is the set of* final states.

Also, $|A|_Q := |Q|$ *and* $|A|_\delta := \sum_{q_1, q_2 \in Q, \alpha \in \Sigma^\varepsilon} \delta(q_1, \alpha, q_2)$ *is the state and transition size respectively.*

Definition 2. *For an NFA* $A = (Q, \Sigma, q_0, \delta, F)$ *and* $w \in \Sigma^*$*, a* run *for* w *is a string* $r = s_0\alpha_1(j_1)s_1 \cdots s_{n-1}\alpha_n(j_n)s_n$*, with* $s_0 = q_0$*,* $s_i \in Q$ *and* $\alpha_i \in \Sigma^\varepsilon$ *such that* $\delta(s_i, \alpha_{i+1}, s_{i+1}) \geq j_{i+1}$ *for* $0 \leq i < n$*, and* $\pi_\Sigma(r) = w$*. A run is accepting if* $s_n \in F$*. The language accepted by* A*, denoted by* $\mathcal{L}(A)$*, is the subset* $\{\pi_\Sigma(r) \mid r$ *is an accepting run in* $A\}$ *of* Σ^**.*

Definition 3 ([7]). *A* prioritized non-deterministic finite automaton (pNFA) *is a tuple* $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$*, where if* $Q := Q_1 \cup Q_2$*, we have: (i)* Q_1 *and* Q_2 *are disjoint finite sets of* states; *(ii)* Σ *is the* input alphabet; *(iii)* $q_0 \in Q$ *is the* initial state; *(iv)* $\delta_1 : Q_1 \times \Sigma \rightarrow Q$ *is the* deterministic*, but not necessarily total,* transition function; *(v)* $\delta_2 : Q_2 \rightarrow Q^*$ *is the* non-deterministic prioritized transition function; *and (vi)* $F \subseteq Q_1$ *are the* final states.

Given a pNFA A, nfa(A) denotes the NFA *associated* with A, which is obtained by ignoring the priorities of the δ_2 transitions of A. Thus for nfa(A), $\delta(p, a, q) = 1$ if $\delta_1(p, a) = q$ for $p \in Q_1$ and $a \in \Sigma$, and $\delta(p, \varepsilon, q) = j$ for $p \in Q_2$, if $\delta_2(p) = q_1 \ldots q_n$, and q appears $j > 0$ times in the sequence $q_1 \ldots q_n$.

Definition 4 ([7]). *For a pNFA* $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$*, a* path *of* $w \in \Sigma^*$ *in* A*, is a run* $s_0\alpha_1(j_1)s_1 \cdots s_{n-1}\alpha_n(j_n)s_n$ *of* w *in nfa*(A)*, such that if* $\alpha_i = \alpha_{i+1} = \ldots = \alpha_{m-1} = \alpha_m = \varepsilon$*, with* $i \leq m$*, then* $(s_{k-1}, j_k, s_k) =$

(s_{l-1}, j_l, s_l), *with* $i \leq k, l \leq m$, *implies* $k = l$ – *i.e. a path is not allowed to repeat the same transition in a sequence of* ε*-transitions. For two paths* $p = s_0\alpha_1(j_1)s_1 \cdots s_{n-1}\alpha_n(j_n)s_n$ *and* $p' = s'_0\alpha'_1(j'_1)s'_1 \cdots s'_{m-1}\alpha'_m(j'_m)s'_m$ *we say that* p *is of higher priority than* p', $p > p'$, *if* $p \neq p'$, $\pi_\Sigma(p) = \pi_\Sigma(p')$ *and either* p' *is a proper prefix of* p, *or if* k *is the first index such that* $(j_k)s_k \neq (j'_k)s'_k$, *then* $\delta_2(s_{k-1}) = \cdots s_k \cdots s'_k \cdots$ *if* $s_k \neq s'_k$, *or* $s_k = s'_k$ *and* $j_k < j'_k$. *An accepting run for* A *on* w *is the highest-priority path* $p = s_0\alpha_1(j_1)s_1 \cdots \alpha_n(j_n)s_n$ *such that* $\pi_\Sigma(p) = w$ *and* $s_n \in F$. *The language accepted by* A, *denoted by* $\mathcal{L}(A)$, *is the subset of* Σ^* *defined by* $\{\pi_\Sigma(r) \mid r \text{ is an accepting run in } A\}$. *Note that* $\mathcal{L}(A) = \mathcal{L}(nfa(A))$.

Infinite loops are avoided in backtracking matchers by disallowing the repetition of the same transition in a sequence of ε-transitions, as specified in the definition above. In [6], the input directed depth-first search algorithm, typically used by backtracking regex matchers to find accepting runs in pNFA, was given, and it was observed that the running time of this algorithm can be described by the size of the backtracking run, defined next. It should be noted that although $\mathcal{L}(A) = \mathcal{L}(\text{nfa}(A))$ for a pNFA A, the purpose of a pNFA is to associate a run deterministic NFA (i.e. an input string can have at most one accepting run) to a regex, and thus to make it possible to define regex extensions such as capturing groups [7].

Definition 5 ([6]). *Let* $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ *be a pNFA,* $q \in Q_1 \cup Q_2$, $w = \alpha_1 \cdots \alpha_n \in \Sigma^*$, *and* $C\colon Q_2 \to \mathbb{N} \cup \{0\}$. *Then the* (q, w, C)-backtracking run *of* A *is a tree over* $Q_1 \cup Q_2 \cup \{Acc, Rej\}$ *(*$Acc, Rej \notin Q_1 \cup Q_2$*). It* succeeds *if and only if* Acc *occurs in it. We denote the* (q, w, C)*-backtracking run by* $btr_A(q, w, C)$ *and inductively define it as follows. If* $q \in F$ *and* $w = \varepsilon$ *then* $btr_A(q, w, C) = q[Acc]$. *Otherwise, we distinguish between two cases:*

- *If* $q \in Q_1$, *then* $btr_A(q, w, C)$ *equals*

$$\begin{cases} q[btr_A(\delta_1(q, \alpha_1), \alpha_2 \cdots \alpha_n, 0^{Q_2})] & \text{if } n > 0 \text{ and } \delta_1(q, \alpha_1) \text{ is defined,} \\ q[Rej] & \text{otherwise.} \end{cases}$$

- *If* $q \in Q_2$ *with* $\delta_2(q) = q_1 \cdots q_k$, *let* $i_0 = C(q) + 1$ *and* $r_i = btr_A(q_i, w, C_{q \mapsto i})$ *for* $i_0 \leq i \leq k$. *Then* $btr_A(q, w, C)$ *equals*

$$\begin{cases} q[Rej] & \text{if } i_0 > k, \\ q[r_{i_0}, \ldots, r_k] & \text{if } i_0 \leq k \text{ but no } r_i \ (i_0 \leq i \leq k) \text{ succeeds,} \\ q[r_{i_0}, \ldots, r_i] & \text{if } i \in \{i_0, \ldots, k\} \text{ is the least index such that } r_i \text{ succeeds.} \end{cases}$$

The backtracking run of A on w *is* $btr_A(w) = btr_A(q_0, w, 0^{Q_2})$. *If* $btr_A(w)$ *succeeds, then the* accepting run of A on w *contains the sequence of states on the right-most path in* $btr_A(w)$.

It should be noted that the argument C, in the definition of $btr_A(q, w, C)$, enforces the condition that a path is not allowed to repeat the same transition in a sequence of ε-transitions in Definition 4. For pNFA without ε-loops, the argument C and corresponding conditions can be removed from the definition of $btr_A(q, w, C)$.

Definition 6. *For a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, the matching time of an input string w with A, is defined to be $|btr_A(w)|$. Let $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*, |w| \leq n\}$ for all $n \in \mathbb{N}$. We say that A has* exponential worst-case matching time *if $f \in 2^{\Omega(n)}$ (or equivalently, if $f(n) \in 2^{\Theta(n)}$) and* polynomial matching time of degree k, *for $k \in \mathbb{N} \cup \{0\}$, if $f \in \Theta(n^k)$.*

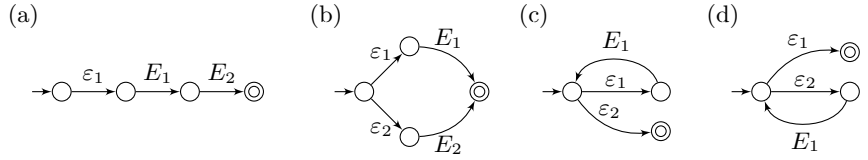

Fig. 1. pNFA corresponding to (a) $E_1 \cdot E_2$, (b) $E_1 \,|\, E_2$, (c) E_1^* and (d) $E_1^{*?}$

We use a regex to pNFA construction, similar to the one implicitly used in the Java regex matching engine. We denote this pNFA constructed from the regex E, by using inductively the constructions in Figure 1 (described in [6]), by $J^p(E)$. In Figure 1(d), $E_1^{*?}$ denotes the reluctant Kleene star operator applied to E_1, which match as few input symbols as possible with E_1, in contrast to greedy Kleene star in Figure 1(c), which matches as many as possible with E_1. Recall that the subscripts of ε indicates the priority of the transition, with ε_1 having the highest priority.

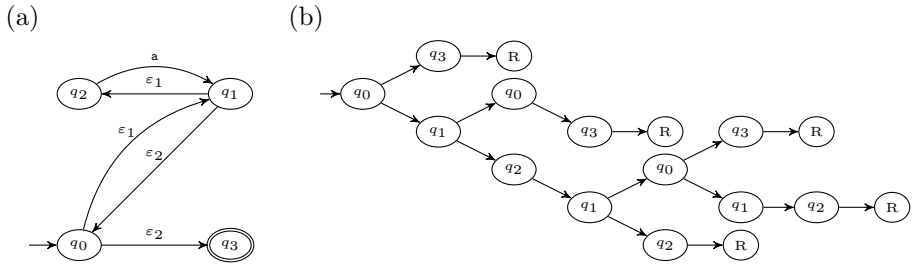

Fig. 2. (a) $J^p((\mathtt{a}^*)^*)$, i.e. the Java pNFA for the regex $(\mathtt{a}^*)^*$. (b) The backtracking run of $J^p((\mathtt{a}^*)^*)$ on input `ax`. The tree is rotated anticlockwise by 90° and highest priority paths are at the bottom. Leaves are marked with `R` instead of `Rej`.

The degree of ambiguity for $w \in \Sigma^*$, with respect to the NFA A, is the number of accepting runs for w in A. The degree of ambiguity of A is the maximum degree of ambiguity over all $w \in \Sigma^*$, which might be infinite, in which case we say A has *infinite degree of ambiguity* (IDA). When A has IDA, we consider the rate at which the maximum number of accepting runs grow in proportion to the length of the input strings, which might be exponential, described by saying A has *exponential degree of ambiguity* (EDA), or polynomial, described as A being *polynomially ambiguous*. Since we determine worst-case matching time of a regex E by using the type of ambiguity of an NFA related to E in a way described in Section 3, the next result is of importance to us.

Theorem 1 ([5]). *Let A be a trim ε-loop free finite automaton. Then*

- *It is decidable in time $O((|A|_\delta + |A|_Q^2)^3)$ whether A is infinitely ambiguous, and in time $O(|A|_\delta^2)$ whether A is exponentially ambiguous.*
- *If A is polynomially ambiguous, the degree of polynomial ambiguity of A can be computed in $O(|A|_\delta^3)$.*

3 Deciding Worst-Case Matching Time

We start this section by defining for a pNFA A, potentially with ε-loops, a pNFA $\text{flat}(A)$, with matching time behavior very similar to that of A, but without ε-loops. To use ambiguity of NFA to analyze worst-case matching time, we first have to remove ε-loops from an pNFA associated to a regex, and this is the main purpose of defining $\text{flat}(A)$. For a pNFA A, let $r_A(Q_2)$ be the subset of Q_2 defined by $Q_2 \cap (\{q_0\} \cup \{\delta_1(q, \alpha) \mid q \in Q_1, \alpha \in \Sigma\})$, i.e. all Q_2 states reachable from a Q_1 state in one transition, and possibly also q_0. A sequence $p_1 j_2 p_2 \cdots p_{n-1} j_n p_n$, with $p_1 \in r_A(Q_2), p_2, \ldots, p_{n-1} \in Q_2, p_n \in Q_1, j_i \in \mathbb{N}$, is a *$\delta_2$-path* if $\delta_2(p_i) = \cdots p_{i+1} \cdots$, $\delta_2(p_i)$ has at least j_{i+1} occurrences of p_{i+1}, and $(p_i, j_{i+1}, p_{i+1}) = (p_k, j_{k+1}, p_{k+1})$ only if $i = k$. Thus δ_2-paths are maximum length subsequences of ε-transitions only, obtained from paths in a pNFA. For a pNFA A, we define a pNFA $\text{flat}(A)$ next, such that the paths for $\text{flat}(A)$ are obtained from those for A, by replacing each δ_2-path with a single ε-transition. This ensures that A and $\text{flat}(A)$ have the same matching time behavior up to a constant.

Definition 7. *For δ_2-paths $P := p_1 j_2 \cdots j_n p_n$ and $P' := p'_1 j'_2 \cdots j'_m p'_m$, with $p_1 = p'_1$, we define $P > P'$ if the least i such that $j_i p_i \neq j'_i p'_i$ is such that $\delta_2(p_{i-1}) = \cdots p_i \cdots p'_i \cdots$ with $p_i \neq p'_i$, or $p_i = p'_i$ but $j_i < j'_i$. We let flat(A) be $(Q_1, r_T(Q_2), \Sigma, q_0, \delta_1, \delta'_2, F)$, where δ'_2 is defined as follows. For $q \in Q'_2$, let $P_1, \ldots, P_n$ be all δ_2-paths, ordered according to priority, starting at q and ending at a state in Q_1. Then $\delta'_2(q) := q_1 \cdots q_n$, where q_i is the last state in P_i.*

An example of going from a pNFA A to $\text{flat}(A)$, is given in Figure 3.

We now describe two algorithms to analyze worst case matching time of regexes. Due to space limitations, these algorithms are not described in-depth. *Simple analysis* is a procedure for determining an upper bound for the worst-case matching time of a regex. We start with a regex E and turn it into $J^p(E)$, the Java version of the pNFA for E. Next, we remove ε-loops by going from $J^p(E)$ to $J' := \text{flat}(J^p(E))$, and then consider the NFA $N := \text{nfa}(J')$. Finally, the NFA N' is obtained from N by adding an additional sink accept state z to N. We place incoming ε-transitions from all other states to z, and make z the only accept state. Going from N to N', turns the problem of counting all possible transitions that can be taken while attempting to match $w \in \Sigma^*$ with N, into counting the number of accepting paths in N' for w. Thus for a given input string w, the size of the backtracking run of w in J' is bounded by the number of accepting paths of w in N', and we have equality if $w \notin \mathcal{L}(N)$. Thus if $w' \in \Sigma^*$ exists such that $ww' \notin \mathcal{L}(N)$ for all $w \in \Sigma^*$, then the worst-case matching time of J' and thus

J, is precisely the ambiguity of N', otherwise the ambiguity for N' is only an upper bound for the worst-case matching time of E.

The unprioritized pNFA (upNFA), up(A), is an NFA obtained from the pNFA A, by not simply ignoring priorities of ε-transitions of A, but doing a type of subset construction that keeps in a given state also track of the states that are reachable with higher priority paths (on the same input). In the construction of up(A), we assume A has no ε-loops, otherwise replace A by flat(A). For an NFA B and Q' a subset of the states of B, let $\overline{Q'}$ be the ε-closure of Q'. For a pNFA A, up(A) is defined next.

Definition 8. *Let $A := (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, then up($A$) is the NFA given by $(Q', \Sigma, q_0', \delta', F')$, where:*

(i) $Q' = ((Q_1 \cup Q_2) \times \mathcal{P}(Q_1)) \setminus Q''$, *where Q'' is the set of states (p, P) such that for all $w \in \Sigma^*$, there is a $p' \in P$, such that w has an accepting path in nfa(A) starting at p';*

(ii) $q_0' = \{(q_0, \emptyset)\}$ *and* $F' = (F \times \mathcal{P}(Q_1)) \cap Q'$;

(iii) *for* $a \in \Sigma$, $\delta'((p, P), a, (p', P')) = 1$ *if* $\delta_1(p, a) = p'$ *and* $\overline{\delta_1(P, a)} \cap Q_1 = P'$, *where δ_1 is extended to be defined on sets of states in the obvious way;*

(iv) $\delta'((p, P), \varepsilon, (p_i, \overline{P \cup \{p_1, \ldots, p_{i-1}\}} \cap Q_1)) = i_j$, *for* $1 \leq i \leq n$, *if* $p \in Q_2$ *and* $\delta_2(p) = p_1 \ldots p_n$, *where i_j is the number of indices i' with $p_i = p_{i'}$ and* $\overline{P \cup \{p_1, \ldots, p_{i-1}\}} \cap Q_1 = \overline{P \cup \{p_1, \ldots, p_{i'-1}\}} \cap Q_1$.

Note that the states of up(A) are of the form (p, P), with $p \in Q$ and $P \subseteq Q_1$. The states from P in (p, P) are those reachable with higher priority paths on the same input. By removing states (p, P) such that for all $w \in \Sigma^*$, there is a $p' \in P$, such that w has an accepting path in nfa(A) starting at p', we ensure that only paths explored in A on any input w is kept in up(A). Just as in our simple analysis, we add a sink accept state z to up(A) to obtain an NFA B', and then perform ambiguity analysis on B'. We refer to the process of constructing $B :=$ up(flat($J^p(E)$)) from a regex E, adding the sink accept state z to B to get B', and then determining the ambiguity of B', as doing a *full analysis* of E. At the cost of going form polynomial to exponential time (in the size of the regular expression), full analysis provides a precise answer.

Example 1. Next we consider the regex E given by `.*|(a*)*`, which has EDA, but only linear worst-case matching time, although the subexpression `(a*)*` has exponential worst-case matching time (with input `a...ab`). However, due to priorities, only the subexpression `.*` is used during matching. The main steps taken in our full analysis are shown in Figures 3 and 4. Since up(flat($J^p(E)$)) has constant ambiguity, E has linear worst-case matching time.

4 Experimental Results

All experiments were performed on a machine with a 3.1GHz, 4 cores and a cache size of 6144KB. We performed simple analysis on the Snort rule-set version 2.9.31

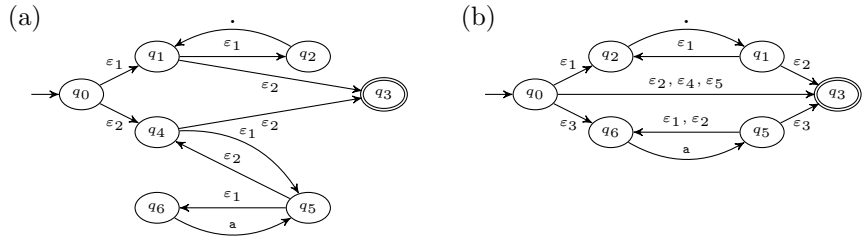

Fig. 3. (a) $J^p(.^* \mid (\mathtt{a}^*)^*)$ and (b) flat$(J^p(.^* \mid (\mathtt{a}^*)^*))$

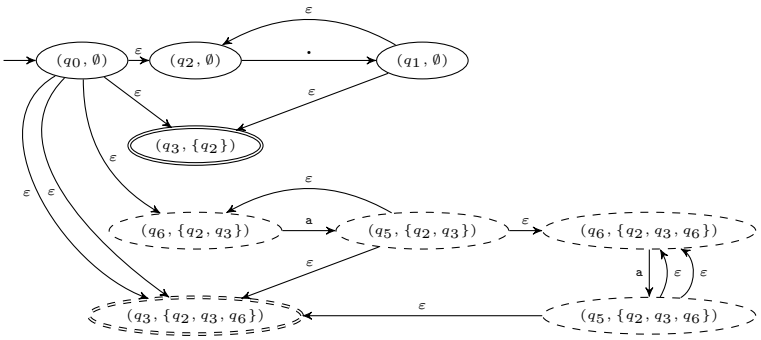

Fig. 4. The unprioritized pNFA for $J^p(.^* \mid (\mathtt{a}^*)^*)$. Dashed states indicate the states Q'' in Definition 8, that should be removed.

([3]; 12499 expressions) and RegExLib ([1]; 2994 expressions) repositories. Simple analysis only checks for EDA and IDA, yielding one of six results: EDA; IDA (but not EDA); No IDA; whether the regex contains illegal syntax or requires unhandled functionality (indicated as "Skipped"); or if the analysis takes an inordinate (10 seconds in our experiments) amount of time (indicated as "Timeout in EDA", or "Timeout in IDA"). Both analyses construct exploit strings with properties, as explained in the Introduction. If EDA or IDA is present in simple analysis, full analysis is performed to determine whether a regex indeed has exponential or non-linear polynomial matching time (although this is strictly speaking only necessary in cases where the exploit strings obtained in simple analysis do not point out the expected behavior). Simple analysis determined that, in total, 156 regexes have EDA, 1041 have IDA, and 9998 have neither. The remaining 4298 regexes were either skipped, or timed out. Full analysis was performed on the 156 and 1041 regexes with EDA and IDA (in simple analysis), respectively. The results of the full analysis is shown in Table 2, which shows whether the matching time of a regex is exponential, polynomial or linear; or whether the analysis timed out. All exponentially vulnerable regexes were tested against the Java regex matcher with their respective exploit strings, as generated by the full analysis. All but two of these regexes did indeed exhibit exponential matching time. The reason for the full analysis producing faulty exploit strings in these two cases warrants further investigation [2].

As mentioned before, if an NFA for a regex contains EDA, it does not necessarily imply that the regex is vulnerable. The regex `(\&d[0-9]{2}=.*?)+`

Repository	EDA	IDA	No IDA	Skipped	Timeout in EDA	Timeout in IDA
Snort	11	824	8381	3108	103	72
RegExLib	145	217	1617	912	16	87

Table 1. A breakdown of the simple analysis results.

Simple Analysis	Full Analysis				
EDA	Exponential	Polynomial	Linear	Timeout in EDA	Timeout in IDA
156	122	0	2	32	0
IDA	Exponential	Polynomial	Linear	Timeout in EDA	Timeout in IDA
1041	0	692	24	0	325

Table 2. The matching time behavior, as determined by full analysis, of the 156 and 1041 regexes shown to have EDA and IDA, respectively, by simple analysis.

from the Snort repository match any input string with a prefix starting with `&d`, followed by two digits and an equals sign. In order to build an exploit string, we can use ε as prefix and `&d00=&d00=` as pump. Since `.*?` matches all strings, two copies of the string `&d00=` can be matched in two ways – either once with the `\&d[0-9]{2}=` and once with the `.*?`, or twice with the `+` operator. But every time the matcher can not match part of the input string with `(\&d[0-9]{2}=)`, it will backtrack and match one character with `.*?`, and thus all strings will be matched in linear time. In the simple analysis, the analyzer detected that the (NFA of the) regex has EDA, but when the full analysis is performed, the analyzer detected that it cannot construct a valid exploit string and therefore classified the regex as not being vulnerable.

Regexes with large constant matching time, might also be regarded as being vulnerable (at least from a practical point of view). Next, we describe an approach that worked well in practice to identify some these regexes. If a regex has a large counted closure, such as $R \coloneqq (S \mid T)\{0, n\}$, for large n, the regex can be approximated (in terms of language accepted and matching time) with a Kleene star, as in $(S \mid T)^*$. The Snort repository contains an expression of this form, namely `\x20\x00([^\x00].|.[^\x00]){255}`. Although the counted closure is of the form $\{n\}$, and not $\{0, n\}$, we can still approximate the regex with `\x20\x00([^\x00].|.[^\x00])+` for the purpose of approximating worst-case matching time. By using this approximation approach, our analysis was able to point out that this regex is indeed vulnerable with the exploit string `\x20\x00` as prefix, `aa` as pump and `\x00\x00` as suffix.

Our analyzer does not yet support all extensions found in Java regexes. Unsupported extensions include possessive quantifiers and back references.

5 Conclusions and Future Work

We developed an analyzer to identify regexes vulnerable to ReDoS. The analysis was run on two repositories of commonly used regexes, and numerous regexes with non-linear worst-case matching time were discovered. We plan to extend our analysis so that most features found in extended regexes are supported, and also to develop techniques to identify regexes with non-constant worst-case

memory usage. One interesting extension to consider is that of possessive quantifiers, allowing the matcher to throw away certain backtracking positions, and creating the possibility to remove some matching time vulnerabilities. To analyze regexes with possessive quantifiers in terms of time and space, we plan to describe the matching of an input string by a pNFA, in terms of a 2-way deterministic pushdown automaton [9]. A further goal is to investigate the complexity of our worst-case matching-time analysis techniques as discussed in Section 3.

References

1. Regexlib. http://www.regxlib.com, accessed: 2015-10-06
2. Regular expression static analysis project page. http://www.cs.sun.ac.za/~abvdm/regex.html, accessed: 2016-04-30
3. Snort. http://www.snort.org, accessed: 2015-10-06
4. Adam, B.: Regular expression dos and node.js. https://blog.liftsecurity.io/2014/11/03/regular-expression-dos-and-node.js (2014)
5. Allauzen, C., Mohri, M., Rastogi, A.: General algorithms for testing the ambiguity of finite automata. In: Ito, M., Toyama, M. (eds.) Developments in Language Theory, 12th International Conference, DLT 2008, Kyoto, Japan, September 16-19, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5257, pp. 108–120. Springer (2008), http://dx.doi.org/10.1007/978-3-540-85780-8_8
6. Berglund, M., Drewes, F., van der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching. In: Ésik, Z., Fülöp, Z. (eds.) Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014. EPTCS, vol. 151, pp. 109–123 (2014), http://dx.doi.org/10.4204/EPTCS.151.7
7. Berglund, M., van der Merwe, B.: On the semantics of regular expression parsing in the wild. In: Drewes, F. (ed.) Implementation and Application of Automata - 20th International Conference, CIAA 2015, Umeå, Sweden, August 18-21, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9223, pp. 292–304. Springer (2015), http://dx.doi.org/10.1007/978-3-319-22360-5_24
8. Cox, R.: Implementing regular expressions (2007), http://swtch.com/~rsc/regexp/, accessed February 26, 2016
9. Gray, J., Harrison, M.A., Ibarra, O.H.: Two-way pushdown automata. Information and Control 11(1/2), 30–70 (1967), http://dx.doi.org/10.1016/S0019-9958(67)90369-5
10. Rathnayake, A., Thielecke, H.: Static analysis for regular expression exponential runtime via substructural logics. CoRR abs/1405.7058 (2014), http://arxiv.org/abs/1405.7058