On DoS Vulnerability of Regular Expressions, with and without Backreferences

Tachio Terauchi
Waseda University
Tokyo, Japan
Email: terauchi@waseda.jp

Abstract*—A regular expression (regex) is said to be vulnerable to the *regex denial of service* (ReDoS) attack if the worst-case running time of a matching algorithm on the regex is super-linear in the length of the input string. Due to the wide-spread usage of regexes, ReDoS is well recognized to be a serious security threat. Meanwhile, *backreference* is an extension to regexes that allows preceding substrings to be used later. The extension is practically popular, supported by many regex engines including those in the standard libraries of Java, Python, JavaScript, and more, and is also known to possess interesting theoretical properties such as the language class of regexes extended with it being outside of that of context-free languages but included in that of indexed languages. This paper is a formal study of ReDoS for regexes with and without backreferences. We make the following contributions: (1) we give a sufficient condition for ReDoS invulnerability in terms of the *degree of ambiguity* of the non-deterministic automaton corresponding to the given regex, using the *memory automata* model of Schmid for the case with backreferences, and (2) we show a transformation method based on state elimination that converts a deterministic memory (or ordinary) automaton to an equivalent ReDoS-invulnerable regex with (or without) backreferences. A corollary of (2) is that, in the case without backreferences, every regex can be converted to an equivalent ReDoS-invulnerable form. Finally, we show that, in stark contrast to the case without backreferences, (3) assuming a certain well-believed conjecture in parameterized complexity theory, there exists no algorithm for converting every regex with backreferences to an equivalent ReDoS-invulnerable form. We note that the positive results (1) and (2) apply to the practically popular but inefficient backtracking matching algorithm, whereas the negative result (3) applies to any matching algorithm.

I. Introduction

Regular expressions (*regexes* for short) have seen wide spread use in modern software development. Many popular programming languages, including Python, Java, JavaScript, and more, have regex engines in their standard libraries with which programs can perform various string-related operations such as sanitizing user inputs [23], [37], [13] and extracting data from texts [5], [24], [12]. Many popular regex engines employ some form of *backtracking matching algorithm* and are vulnerable to the *regex denial of service* (ReDoS) attack which exploits the complexity of the algorithm. Formally, a regex is said to be ReDoS vulnerable if the worst-case running time of the matching algorithm on the regex is super-linear in the length of the input string. Due to the wide spread use of regexes, ReDoS is a serious threat to the security of software systems [15], [1].

Meanwhile, *backreference* is an extension to regexes that allows preceding substrings to be used later. It is broadly known that backreference is a practical extension and is supported by many popular regex engines including those in the standard libraries of Java, Python, and JavaScript. Furthermore, regexes extended with backreferences (*rewbs* for short) are known to possess interesting theoretical properties, such as their expressive power being no longer regular (in fact, not even context-free) [6] and properly contained in the class of indexed languages [26]. Schmid has proposed a class of automata called *memory automata* (MFA) that corresponds to the language class of rewbs [28].

We conduct a formal study on ReDoS for regexes with and without backreference, and make the following contributions:

(1) A definition of the degree of ambiguity for MFA that naturally extends that for NFA, and a sufficient condition for ReDoS invulnerability for regexes with (and without) backreferences in terms of the measure (Section III);
(2) An algorithm that converts a deterministic memory (resp. ordinary) automaton to an equivalent ReDoS-invulnerable regex with (resp. without) backreferences (Section IV);[1] and
(3) A result that, assuming a certain well-believed conjecture in parameterized complexity theory, there can be no algorithm for converting an arbitrary rewb to an equivalent ReDoS-invulnerable form (Section V).

Since any pure regex (i.e., one not containing a backreference) can be converted to an equivalent DFA, result (2) implies that it is possible to convert an arbitrary pure regex to an equivalent ReDoS invulnerable form. Thus, result (3) shows that the situation is rather different when backreferences are allowed. We remark that the positive results (1) and (2) apply to the practically popular but inefficient backtracking matching algorithm, whereas the negative result (3) applies to any matching algorithm.

Considering backreferences in a study on ReDoS is important especially because, despite their wide-spread use in practice, recent ReDoS-free non-backtracking matching algorithms such as Google's RE2 and those proposed in very recent papers [25], [21], [4] have been dropping the support for backreferences in order to ensure ReDoS freedom. Our work

[1]The algorithm can be easily extended to take a finite union of deterministic automata (cf. Section IV).

sheds light on how to provide some support for backreferences while still avoiding ReDoS. Namely, result (1) gives a novel sufficient condition for rewbs to be ReDoS invulnerable, while result (2) gives a way to construct such rewbs in some cases, and result (3) gives a limit on doing such a construction in general.

The rest of the paper is organized as follows. Section II defines preliminary notions. Sections III, IV, and V contain the main results mentioned above. Section VI discusses related work. Section VII concludes the paper with a discussion on future work. For space, some proofs are deferred to the appendix.

II. Preliminaries

$\mathbb{N}$ is the set of non-negative integers, and $\mathbb{N}_{\geq 1} = \{i \in \mathbb{N} \mid i \geq 1\}$. We assume a finite alphabet Σ. A *string* is a finite sequence of characters in Σ. We write Σ_ϵ for $\Sigma \cup \{\epsilon\}$ where ϵ is the empty string. We write $w_1 \cdot w_2$ (or simply $w_1 w_2$ when clear) for the string obtained by concatenating strings w_1 and w_2. We write $|w|$ for the length of the string. We write $w \preceq w'$ if w is a prefix of w', and $w \prec w'$ especially when it is a proper prefix. For $w \preceq w'$, we write $w \backslash w'$ for the left quotient of w' with respect to w, that is, $w \backslash w' = u$ such that $wu = w'$. For a map f, we write $f[a \mapsto b]$ for the map $\lambda x.\texttt{if } x = a \texttt{ then } b \texttt{ else } f(x)$. We write $S \uplus S'$ for the disjoint union of sets S and S', i.e., $S \uplus S' = S \cup S'$ if $S \cap S' = \emptyset$ and is undefined otherwise.

The syntax of a *regex with backreferences* (*rewb* for short) is given by the following grammar:

$$r ::= \ell \mid \emptyset \mid r_1 \cdot r_2 \mid r_1 | r_2 \mid r^* \mid ({}_i r)_i \mid \backslash i$$

Here, $\ell \in \Sigma_\epsilon$, $\emptyset$ is empty, $r_1 \cdot r_2$ is concatenation, $r_1 | r_2$ is union, and r^* is repetition (i.e., *Kleene star*), $({}_i r)_i$ is a capturing group with the index $i \in \mathbb{N}_{\geq 1}$, and $\backslash i$ is a backreference, with the requirement that in $({}_i r)_i$, r does not contain a capturing group or a backreference of the same index i. The precedence of the operators is as follows, from higher to lower: repetition, concatenation, and union. We sometimes write $r_1 \cdot r_2$ as $r_1 r_2$. To clarify, we sometimes call a rewb containing no capturing groups and backreferences a *pure regex* or a *regex without backreferences*. We describe the semantics of rewbs informally (a formal semantics will be given later by a translation to MFAs): a backreference $\backslash i$ matches a string that was matched by the nearest capturing group $({}_i r)_i$, where strings matched by a capturing group $({}_i r)_i$ are those matched by r. The behavior of the other operators (i.e., empty, concatenation, union, and repetition) are the same as those for pure regexes. The *language* of a rewb r, $L(r)$, is defined to be the set of strings matched by r. We say that rewbs r and r' are *language equivalent* (or just *equivalent*) when $L(r) = L(r')$.

Example II.1. Let $\Sigma = \{a, b\}$ be the alphabet. The language of the rewb $r = ({}_1(a|b)^*)_1 \backslash 1$ is $L(r) = \{ww \mid w \in \Sigma^*\}$.

Because this $L(r)$ is a textbook example of a non-context-free language (and therefore, non-regular), the expressive power of rewbs exceeds that of the pure ones. Additionally, the expressive power of rewbs is known not to contain that of context-free languages (and therefore, because of the above, incomparable to that of context-free languages) [9], [6], and a recent work has shown that it is properly contained in that of indexed languages [26].

Next, we recall a class of automata called *memory automata* (MFA) that was introduced by Schmid [28]. Our definition is roughly equivalent to Schmid's but with some simplifications on status change operations (called *memory instructions* in [28]) for brevity, and an extension to allow matching only a proper prefix of a memory content at a backreference operation. We explain the details of the latter change when describing the transitions of MFAs. We will also use MFAs to give a formal semantics of rewbs.

Formally, an MFA is a tuple (Q, δ, q_0, F) where Q is a finite set of states, $\delta : (Q \times (\Sigma_\epsilon \uplus \mathit{BrOps}) \times \mathit{ScOps}) \to \mathcal{P}(Q)$ is a finite function where $\mathit{BrOps} = \{\backslash \boldsymbol{i} \mid i \in \mathbb{N}_{\geq 1}\}$ is the set of *backreference operations*, $\mathit{ScOps} = \{\diamond, (_{\boldsymbol{i}},)_{\boldsymbol{i}} \mid i \in \mathbb{N}_{\geq 1}\}$ is the set of *status change operations*, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A configuration of the MFA is a tuple (q, σ) where $q \in Q$ and $\sigma : \mathbb{N}_{\geq 1} \to \Sigma^* \times \{\mathcal{O}, \mathcal{C}\}$. For each $i \in \mathbb{N}_{\geq 1}$, we call $\sigma(i)$ a *memory* where the first element is the *content* of the memory and the second element is the *status* of the memory. A status is either *open* ($\mathcal{O}$) or *closed* ($\mathcal{C}$). For $(x, s) = \sigma(i)$, we write $\sigma(i).\mathit{con}$ for x and $\sigma(i).\mathit{sta}$ for s. Roughly, $\delta(q, \alpha, s) \ni q'$ means that, from the state q, the automaton can transition to the state q' by either consuming an input symbol matching α if $\alpha \in \Sigma$, consuming no input symbols if $\alpha = \epsilon$, or consuming input symbols matching the string $\sigma(i).\mathit{con}$ if $\alpha = \backslash \boldsymbol{i}$ where σ is the current memories. Also, an open status means that the memory is open for writing and closed status means that that the writing has finished and the memory is ready to be read. More specifically, at each transition step the memories are updated so that their statuses are updated according to the status change operation s, and their contents are updated so that those of open status are appended with the consumed sequence of input symbols and those of closed status remain unchanged.

We formalize the notion of transitions. For $w \in \Sigma^*$, $s \in \mathit{ScOps}$, and $\sigma : \mathbb{N}_{\geq 1} \to \Sigma^* \times \{\mathcal{O}, \mathcal{C}\}$, we define $\mathit{update}(\sigma, s, w) = \mathit{upd}_{\mathit{con}}(\mathit{upd}_{\mathit{sta}}(\sigma, s), w)$ where

$$\begin{array}{lcl}
\mathit{upd}_{\mathit{sta}}(\sigma, \diamond) & = & \sigma \\
\mathit{upd}_{\mathit{sta}}(\sigma, (_{\boldsymbol{i}})) & = & \sigma[i \mapsto (\sigma(i).\mathit{con}, \mathcal{O})] \text{ if } \sigma(i).\mathit{sta} = \mathcal{C} \\
\mathit{upd}_{\mathit{sta}}(\sigma,)_{\boldsymbol{i}}) & = & \sigma[i \mapsto (\sigma(i).\mathit{con}, \mathcal{C})] \text{ if } \sigma(i).\mathit{sta} = \mathcal{O} \\
\mathit{upd}_{\mathit{con}}(\sigma, w) & = & \{i \mapsto (\sigma(i).\mathit{con} \cdot v_i, \sigma(i).\mathit{sta}) \mid \\
 & & \quad (\sigma(i).\mathit{sta} = \mathcal{O} \wedge v_i = w) \vee \\
 & & \quad (\sigma(i).\mathit{sta} = \mathcal{C} \wedge v_i = \epsilon)\}
\end{array}$$

Here, $\mathit{upd}_{\mathit{sta}}(\sigma, (_{\boldsymbol{i}})$ and $\mathit{upd}_{\mathit{sta}}(\sigma,)_{\boldsymbol{i}})$ are undefined if the respective conditions are not met.

Definition II.2 (One-step transition)**.** A (one-step) *transition*, $(q, \sigma) \xrightarrow{w, \alpha, s} (q', \sigma')$, is defined to the smallest relation satisfying the following:

(1) $\alpha \in \Sigma_\epsilon$, $\delta(q, \alpha, s) \ni q'$, $w = \alpha$, and $\sigma' = update(\sigma, s, w)$;
(2) $\alpha = \backslash \boldsymbol{i} \in BrOps$, $\sigma(i).sta = \mathcal{C}$, $w = \sigma(i).con$, $\delta(q, \alpha, s) \ni q'$ and $\sigma' = update(\sigma, s, w)$; or
(3) $\alpha = \backslash \boldsymbol{i} \in BrOps$, $\sigma(i).sta = \mathcal{C}$, $w \prec \sigma(i).con$, $(q, \backslash \boldsymbol{i}, \_, \_) \in \delta$, stuck-st $= q'$, and $\sigma' = \sigma$.

Case (1) corresponds to ordinary transitions that consume a single character or no characters (i.e., ϵ transitions) but with possible change in the memories as stipulated by the $update$ function. Namely, the consumed character is appended to the contents of open memories and memory statuses are changed according to the status change operator s. Cases (2) and (3) process a backreference operation that reads a memory where case (2) handles the case when the consumed string matches the content of the memory and (3) handles the case when the consumed string only matches a proper prefix of the content. Here, stuck-st $\notin Q$ is a special *stuck* state that indicates that only a proper prefix of the content was matched.[2] The original definition of MFA by Schmid [28] does not have transitions corresponding to (3), but it does not affect the set of words accepted by an MFA because stuck-st is not accepting and also has no outgoing transitions. We make the change because it allows defining the degree of ambiguity to simply be the number of matching paths (cf. Section III). Note that a backreference transition can be taken only when the referred memory is closed. Also, because of the definition of $update$, a transition cannot re-open an already open memory or re-close an already closed memory. We often write $(q, \sigma) \xrightarrow{w,\alpha,s} (q', \sigma')$ as $(q, \sigma) \xrightarrow{w} (q', \sigma')$ when only the consumed string is relevant.

A *path* π is a possibly empty finite sequence of one-step transitions

$$(q_1, \sigma_1) \xrightarrow{w_1} (q_2, \sigma_2) \xrightarrow{w_2} \dots \xrightarrow{w_n} (q_n, \sigma_n)$$

where the length of π, denoted $|\pi|$, is n. We say that π *matches* the string $w_1 w_2 \dots w_n$, and denote the latter by $word(\pi)$. The *initial configuration* is $(q_0, \lambda\_.(\epsilon, \mathcal{C}))$, that is, it is in the initial state q_0 and all of its memories are closed and contain empty strings. A configuration (q, σ) is *accepting* if q is a final state. A path is *initial* if it starts from the initial configuration. We say that the string $w \in \Sigma^*$ is *accepted* by the MFA, if there is an initial path π such that $word(\pi) = w$ and π ends in an accepting configuration. Such a path is called a *run* that accepts w. For an MFA A, we define the *language* of A, $L(A)$, to be the set of strings accepted by A. We say that MFAs A and A' are *equivalent* when $L(A) = L(A')$, and likewise, we say that a MFA A and a rewb r are equivalent when $L(A) = L(r)$.

Example II.3. Figure 1 shows two MFAs. Here, a labeled edge $q \xrightarrow{\alpha/s} q'$ represents a transition (q, α, s, q'), i.e., the transition moves from the state q to q', consumes input symbols according to α, and makes memory status change according to s. Unlabeled edges are $\epsilon/\diamond$ transitions. A_1 is equivalent to the rewb from Example II.1, that is, $L(A_1) = \{ww \mid w \in \{a, b\}^*\}$. A_2 is equivalent to the rewb $({}_1(a|b)^*)_1 c \backslash 1$, and its language is $L(A_2) = \{wcw \mid w \in \{a, b\}^*\}$.

Note that an MFA whose transition relation is included in $Q \times \Sigma_\epsilon \times ScOps \times Q$ can be seen as an ordinary nondeterministic finite automaton (NFA) by ignoring the memories part of configurations.

We say that an MFA is *nested*, if for any path

$$(q_1, \sigma_1) \Rightarrow (q_2, \sigma_2) \Rightarrow \dots \Rightarrow (q_n, \sigma_n)$$

there exist no $1 \le m_1 < m_2 < m_3 < m_4 < n$ and $i, j \in \mathbb{N}_{\ge 1}$ with $i \ne j$, such that (1) the transition from (q_{m_1}, σ_{m_1}) to $(q_{m_1+1}, \sigma_{m_1+1})$ opens the i-th memory, (2) the transition from (q_{m_2}, σ_{m_2}) to $(q_{m_2+1}, \sigma_{m_2+1})$ opens the j-th memory, (3) the transition from (q_{m_3}, σ_{m_3}) to $(q_{m_3+1}, \sigma_{m_3+1})$ closes the i-th memory, and (4) the transition from (q_{m_4}, σ_{m_4}) to $(q_{m_4+1}, \sigma_{m_4+1})$ closes the j-th memory. Intuitively, nested means that that the MFA has syntactically proper capturing groups when it is viewed as a regex. For example, the MFA corresponding to the syntactically improper regex $({}_1a({}_2b)_1c)_2$ violates the condition. As shown in [28], any MFA can be converted to an equivalent nested form.

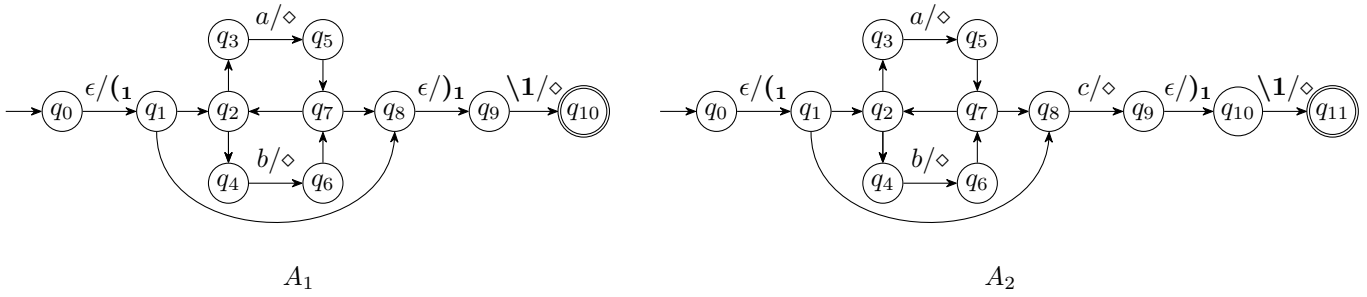
A *deterministic* MFA (DMFA) is an MFA (Q, δ, q_0, F) satisfying: (1) for any $q \in Q$, $\alpha \in \Sigma_\epsilon \cup BrOps$, $s \in ScOps$, $|\delta(q, \alpha, s)| \le 1$, (2) for any $q \in Q$, $\alpha \in \{\epsilon\} \cup BrOps$, $\alpha' \in \Sigma_\epsilon \cup BrOps$, and $s, s' \in ScOps$, if $\alpha \ne \alpha'$ and $\delta(q, \alpha, s) \ne \emptyset$ (in which case $|\delta(q, \alpha, s)| = 1$ by (1)) then $\delta(q, \alpha', s') = \emptyset$, and (3) for any $q \in Q$, $\alpha, \alpha' \in \Sigma_\epsilon \cup BrOps$, and $s, s' \in ScOps$, if $\delta(q, \alpha, s) = \delta(q, \alpha', s') \ne \emptyset$ then $s = s'$. We note that our definition of DMFA is equivalent to that of [28] modulo the modifications that we made on the definition of MFA remarked before.[3] Note that for an MFA whose transition relation is included in $Q \times \Sigma_\epsilon \times ScOps \times Q$ (and thus can be viewed as an NFA), our definition of DMFA coincides with the usual definition of deterministic finite automaton (DFA) except it allows an ϵ transition from a state when the transition is the only transition from the state.

As shown in [28], a rewb can be translated to an equivalent MFA by adopting the well-known Thompson's construction [32]. Formally, the (extended) Thompson's construction applied to a rewb r, $MFA'(r)$, is defined inductively as follows.

(1) For $\ell \in \Sigma_\epsilon$, $MFA'(\ell) = (\{q_0, q_f\}, \delta, q_0, \{q_f\})$ where $\delta = \{(q_0, \ell, \diamond, q_f)\}$ and $q_0 \ne q_f$;
(2) $MFA'(\emptyset) = (\{q_0, q_f\}, \emptyset, q_0, \{q_f\})$ and $q_0 \ne q_f$;
(3) $MFA'(r_1 r_2) = (Q_1 \uplus Q_2, \delta_1 \uplus \delta_2 \uplus \delta, q_{01}, \{q_{f2}\})$ where $MFA'(r_1) = (Q_1, \delta_1, q_{01}, \{q_{f1}\})$, $MFA'(r_2) = (Q_2, \delta_2, q_{02}, \{q_{f2}\})$, and $\delta = \{(q_{f1}, \epsilon, \diamond, q_{02})\}$;
(4) $MFA'(r_1|r_2) = (Q_1 \uplus Q_2 \uplus \{q_0, q_f\}, \delta_1 \uplus \delta_2 \uplus \delta, q_0, \{q_f\})$ where $MFA'(r_1) = (Q_1, \delta_1, q_{01}, \{q_{f1}\})$, $MFA'(r_2) = (Q_2, \delta_2, q_{02}, \{q_{f2}\})$, $\delta = \{(q_0, \epsilon, \diamond, q_{0i}), (q_{fi}, \epsilon, \diamond, q_f) \mid i \in \{1, 2\}\}$ and $q_0 \ne q_f$;

[2] It is similar to the *trap state* of [20].

[3] Technically, DMFA of [28] does not allow ϵ transitions and also uses a weaker version of (3) called *pseudo determinism*, but it is easy to see that both definitions have equivalent expressive powers.



A_1 A_2

Fig. 1. MFA examples.

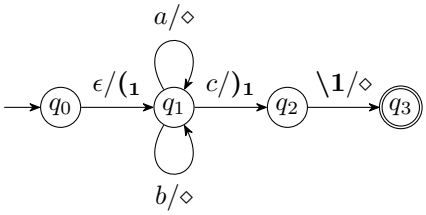


Fig. 2. A DMFA equivalent to $(_1(a|b)^*)_1 c\backslash 1)$.

(5) $MFA'(r^*) = (Q \uplus \{q_0, q_f\}, \delta \uplus \delta', q_0, \{q_f\})$ where $MFA'(r) = (Q, \delta', q_0', \{q_f'\})$, $q_0 \neq q_f$, and $\delta = \{(q_0, \epsilon, \diamond, q_0'), (q_0, \epsilon, \diamond, q_f), (q_f', \epsilon, q_0'), (q_f', \epsilon, q_f)\}$;

(6) $MFA'((_i r)_i) = (Q \uplus \{q_0, q_f\}, \delta \uplus \delta', q_0, \{q_f\})$ where $MFA'(r) = (Q, \delta', q_0', \{q_f'\})$, $q_0 \neq q_f$, and $\delta = \{(q_0, \epsilon, (_i, q_0'), (q_f', \epsilon,)_i, q_f)\}$; and

(7) $MFA'(\backslash i) = (\{q_0, q_f\}, \{(q_0, \backslash i, \diamond, q_f)\}, q_0, \{q_f\})$ where $q_0 \neq q_f$.

For our purposes, it is convenient to *trim* the produced MFA by removing any state that is unreachable from the initial state or cannot reach the final state. We define $MFA(r)$ to be the MFA obtained by trimming $MFA'(r)$. It is easy to see that $MFA(r)$ is nested for any rewb r. We formally define the language accepted by a rewb r, $L(r)$, to be $L(MFA(r))$. Note that $MFA(r)$ is an NFA when r is a pure regex.

Example II.4. Recall the MFAs from Example II.3. It is easy to see that they are obtained by applying the Thompson's construction on the corresponding rewbs, that is, $A_1 = MFA((_1(a|b)^*)_1\backslash 1)$ and $A_2 = MFA((_1(a|b)^*)_1 c\backslash 1)$. Both are nested. Neither are deterministic. Namely, the states q_1, q_2, and q_7 in both automata have multiple out-going ϵ transitions. In fact, as shown by Schmid [28], there is no DMFA equivalent to A_1 (and thus equivalent to $(_1(a|b)^*)_1\backslash 1)$. By contrast, as we show in Figure 2, there is a DMFA that is equivalent to A_2. Note that c acts as a "delimiter" for the automaton to deterministically decide when to transition from q_1 to q_2.

III. Characterization of Vulnerability

In this section, we show that the ReDoS vulnerability of rewbs can be characterized by a measure called the *degree of ambiguity*. We first formalize *backtrack matching algorithm* (BMA). As mentioned in Section I, BMAs are used in most major regex engines in practice, and ReDoS typically exploits the worst-case, oft-called "catastrophic", behavior of BMAs. A BMA is essentially a depth-first search (DFS) algorithm that looks for an accepting path matching the given string in the automaton corresponding to the given regex, or equivalently, a recursive-descent parsing algorithm when viewed as working directly on the expression rather than the automaton corresponding to it [7], [30], [14].

Algorithm 1 Backtracking matching algorithm

Input: A rewb r and a string w
Output: `true` if $w \in L(r)$ and `false` if $w \notin L(r)$

```
1:  (Q, δ, q0, F) ← MFA(r)
2:  procedure IsPrefix(w', w)
3:      if w' = ε then return true
4:      else if ∃a ∈ Σ.w = au ∧ w' = au' then
5:          return IsPrefix(u', u)
6:      else return false
7:  procedure Dfs(q, σ, w)
8:      if w = ε then
9:          if ∃q' ∈ ε-reach(q, σ).q' ∈ F then
10:             return true
11:         else return false
12:     for π ∈ Next(q, σ) do
13:         if IsPrefix(word(π), w) then
14:             (q', σ') ← last(π)
15:             if Dfs(q', σ', word(π)\w) then
16:                 return true
17:     return false
18: return Dfs(q0, λ_.(ε, C), w)
```

Algorithm 1 formally defines the BMA. It first converts the given rewb to an MFA (line 1), and then calls the procedure Dfs (line 18). Here, ϵ-*reach*(q, σ) returns the set of states that is reachable from the configuration (q, σ) by paths matching ϵ. To prevent loops, we require the ϵ-matching path to be cycle-free. Formally, we say that a path π to be *ϵ-cycle free* (ϵcf) if it contains no subsequence of the form $(q, \sigma) \xrightarrow{\epsilon}{}^{+} (q, \sigma)$, where $\xrightarrow{\epsilon}{}^{+}$ denotes one or more ϵ-matching transitions. We define ϵ-*paths*(q, σ) to be the set of ϵcf paths from (q, σ) matching ϵ, that is, it is the set of ϵcf paths of the form $(q, \sigma) \xrightarrow{\epsilon}{}^{*} (q', \sigma')$ where $\xrightarrow{\epsilon}{}^{*}$ denotes zero or more ϵ-matching transitions. Then, ϵ-*reach*$(q, \sigma) = \{q' \mid \exists \pi \in$

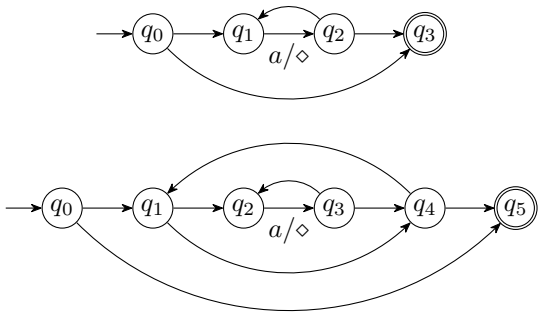


Fig. 3. $MFA(a^*)$ and $MFA(a^{**})$

ϵ-*paths*$(q,\sigma).q'$ is the last state of $\pi\}$.

$Next(q,\sigma)$ is defined to be the set of paths $\{\pi \cdot \pi' \mid \pi \in \epsilon\text{-}paths(q,\sigma) \wedge \pi' = last(\pi) \xrightarrow{w} (q', \_) \wedge w \neq \epsilon \wedge q' \neq \textsf{stuck-st}\}$ where $last(\pi)$ denotes the last configuration of π. That is, it is the set of paths from (q,σ) formed by extending a (possibly empty) ϵcf ϵ-matching path with one non-ϵ-matching transition such that the path does not end in stuck-st. Recall from Section II that stuck-st is the special "stuck" state. Roughly, $\pi \in Next(q,\sigma)$ means that π is a "next" path that can be taken from (q,σ) by making one non-ϵ transition.

DFS(q,σ,w) is a DFS algorithm that explores paths from the input configuration (q,σ), and tries to find one that matches the input string w and reaches an accepting configuration. When w is empty, it checks if there is an accepting configuration ϵ-reachable from (q,σ), and if so returns `true` and otherwise returns `false` (line 9). Otherwise (i.e., when $w \neq \epsilon$), it recursively calls DFS to continue the search in a depth-first manner (line 12). Here, a recursive function ISPREFIX is used to decide if the to-be-consumed string w' can be matched with the remaining string w. Note that ISPREFIX(w', w) returns `true` if $w' \preceq w$ and otherwise returns `false`. Note that the first argument to ISPREFIX will always be a string of length one for pure regexes and so the complexity of ISPREFIX is $O(1)$ for such regexes. However, this is not the case for regexes with backreferences because the second argument can be any string that is stored in a memory.

Note that each recursive call to DFS and ISPREFIX is made on strictly shorter strings. Therefore, Algorithm 1 always converges, making at most $|w|$ deep recursive calls. Furthermore, it is easy to see that DFS(q,σ,w,ϵ) returns `true` iff there is a path from (q,σ) to an accepting configuration that matches w, and otherwise returns `false`. Therefore, the algorithm correctly decides if $w \in L(r)$ or not.

Algorithm 1 can exhibit time complexity that is exponential in the length of the input string because it performs DFS over the paths of $MFA(r)$ that start from the initial configuration and match w until it finds one that reaches a final state. Note that the algorithm has a choice on which branch of the DFS tree to explore next (cf. line 12). We consider the worst-case behavior, as often is the case in security research, to fall on the safe side. Let us write $TIME_{BMA}(r,w)$ for the worst-case running time of Algorithm 1 on the input rewb r and the input string w. Because ReDoS is only concerned with the complexity relative to the input string, it is safe to equate $TIME_{BMA}(r,w)$ with the number of calls to DFS and ISPREFIX. Note, in particular, that the complexity of $Next$ can be considered $O(1)$.

Definition III.1 (ReDoS Vulnerability)**.** A rewb r is said to be *ReDoS vulnerable* if $TIME_{BMA}(r,w)$ is super-linear in $|w|$, that is, if $TIME_{BMA}(r,w) \notin O(|w|)$.

Example III.2. Figure 3 shows $MFA(a^*)$ (above) and $MFA(a^{**})$ (below) both over the alphabet $\{a,b\}$. Neither are deterministic. However, Algorithm 1 is guaranteed to converge in time linear in the length of the input string on the former while it exhibits worst-case exponential time complexity on the latter. For instance, when given strings of the form a^nb as input, where a^n is a repeated n many times, the former converges and returns `false` after making $O(n)$ many DFS calls. By contrast, the latter makes $\Omega(2^n)$ many DFS calls before returning `false`. The latter happens because, each time the algorithm reaches the state q_2 and a is the next character to be matched, it has the choice of returning to q_2 by following the sequence of transitions $q_2 \xrightarrow{a/\diamond} q_3 \to q_2$ or $q_2 \xrightarrow{a/\diamond} q_3 \to q_4 \to q_1 \to q_2$, and the algorithm will end up (in vain) trying both possibilities by backtracking at every position where a occurs in a^nb. Therefore, a^{**} is ReDoS vulnerable whereas a^* is not.

Note that the two expressions in the above example are equivalent, accepting the same language $\{a^n \mid n \in \mathbb{N}\}$. The technique we shall present in Section IV can be used to convert a vulnerable expression like a^{**} to an equivalent invulnerable one like a^*. In fact, it can be used to convert any pure regex into an equivalent invulnerable one.

Our definition of ReDoS vulnerability is consistent with those in the prior studies on ReDoS [31], [36], [14], [11], [27]. In particular, it concerns the time complexity of the matching algorithm over only the input strings and not the input expressions. The convention is followed because the expression is often fixed in practice and only the input string is possibly under the infuence of the adversary. However, it is important to note that the expression does affect the complexity relative to the input string. Indeed, as seen in Example III.2, a BMA can exhibit widely different running times relative to the input string length for the same set of input strings but on different expressions, and even when the expressions denote the same language.

Remark III.3. We elaborate on the reason for the special focus on BMA in our and others' work on ReDoS. BMA is not the only algorithm for deciding regex matching, and is also well known to be not the most efficient algorithm for the task. For example, for pure regexes, Thompson's algorithm from the 1960's, based on breadth-first search (BFS), can decide the regex matching problem in time linear in the length of the input string [32]. BFS is also used in recent ReDoS-free non-backtracking matching algorithms that support some

extensions such as lookarounds (but, importantly, *not* backreferences) [25], [4].

Nonetheless, most existing research on ReDoS focus on BMA because BMA is the most popular algorithm for deciding regex matching that is currently used in practice. In particular, BMA is used in the regex engines of popular programming languages including Java, Python, JavaScript, and more. The reason for the popularity is two-fold. The first reason is ease of implementation. Importantly, popular extensions such as backreferences, substring extractions by captures, and lookarounds can be easily implemented in BMA. The second reason is good empirical (or "average case") complexity. While BMA has terrible worst-case complexity and thus becomes a prime target of ReDoS, it is known to usually work well on ordinary (i.e., "non-malicious") inputs.

We next define the *degree of ambiguity* for MFAs by naturally extending the notion of the same name studied for NFAs [35]. Formally, the *degree of ambiguity* of an MFA A with respect to a string w, $da(A, w)$, is the number of (not necessarily accepting) ϵcf initial paths of A that matches w, that is, $da(A, w) = |\{\pi \in \mathit{InitPaths}(A) \mid \pi \text{ is } \epsilon cf \wedge \mathit{word}(\pi) = w\}|$ where $\mathit{InitPaths}(A)$ is the set of initial paths of A. For a rewb r, we define its degree of ambiguity by $da(r, w) = da(\mathit{MFA}(r), w)$. We write $da(r)$ for $\max_{w \in \Sigma^*} da(r, w)$, letting $da(r) = \infty$ when there is no $k \in \mathbb{N}$ such that $da(r, w) \leq k$ for all $w \in \Sigma^*$.

Definition III.4 (CDA)**.** A rewb r is said to be *constant degree ambiguous* (CDA) if $da(r) \neq \infty$.

Example III.5. The rewb a^* from Example III.2 is CDA over the alphabet $\Sigma = \{a, b\}$ because $da(a^*, w) \leq 2$ for any $w \in \Sigma^*$. To see this, note that $da(a^*, w) = 0$ for any $w \notin L(a^*)$ because there are no paths of $\mathit{MFA}(a^*)$ matching such w, whereas $da(a^*, a^n) = 2$ for any $n \geq 0$ because there are exactly two paths, $q_0 \ (\xrightarrow{\epsilon} q_1 \xrightarrow{a} q_2)^n$ and $q_0 \ (\xrightarrow{\epsilon} q_1 \xrightarrow{a} q_2)^n \xrightarrow{\epsilon} q_3$, that match a^n. By contrast, a^{**} is not CDA over Σ. This is so because, after reaching q_2, for each a, there are two possible sequence of transitions, $q_2 \xrightarrow{a/\diamond} q_3 \to q_2$ or $q_2 \xrightarrow{a/\diamond} q_3 \to q_4 \to q_1 \to q_2$, that can be taken before matching the next a. This gives 2^n many paths that can match a^n. Therefore, $da(a^{**}, a^n) \geq 2^n$, and thus a^{**} is not CDA. Likewise, for the rewbs from Examples II.3 and II.4, one can show that $({}_1(a|b)^*)_1 c \backslash 1$ is CDA whereas $({}_1(a|b)^*)_1 \backslash 1$ is not. Namely, for the latter, $da(({}_1(a|b)^*)_1 \backslash 1, a^{2n}) \geq n$ because there are n many paths, each matching the initial a^{n+m} (for $m \in \{0, \ldots, n\}$) prefix of a^{2n} in the part of $\mathit{MFA}(({}_1(a|b)^*)_1 \backslash 1)$ comprising the states $q_0, \ldots, q_7$ before transitioning to the state q_8 to match the rest.

The first main result of our paper is the following theorem which shows that CDA is a sufficient condition for guaranteeing ReDoS invulnerability.

Theorem III.6. *A rewb is not ReDoS vulnerable if it is CDA.*

Proof. Given an input string w, Algorithm 1 explores the paths of the automaton $\mathit{MFA}(r)$, and in the worst case explores all ϵcf initial paths matching each prefix $w' \preceq w$. The string argument in any call to DFS is a suffix of w, that is, $w' \backslash w$ for some $w' \preceq w$. Then, each call $\textsc{Dfs}(q, \sigma, \mathit{word}(\pi) \backslash w)$ at line 15 can be attributed to a call to IsPrefix at line 13 that immediately precedes it. That is, all calls to DFS except the root one at line 18. Furthermore, for each such call to DFS, each call $\textsc{IsPrefix}(w_1, w_1')$ that precedes it, including the ones called recursively at line 5, can be attributed to a distinct ϵcf initial path matching $u \preceq w$ where $u w_1' = w'$. Therefore, $\mathit{TIME}_{\mathit{BMA}}(r, w) \in O(\sum_{w' \preceq w} da(r, w'))$.

Suppose that r is CDA. Then, there is $k \in \mathbb{N}$ such that $da(r, w) \leq k$ for any $w \in \Sigma^*$. Therefore, by the argument in the previous paragraph, $\mathit{TIME}_{\mathit{BMA}}(r, w) \in O(\sum_{w' \preceq w} k) = O(|w|)$, and so r is not ReDoS vulnerable. □

As remarked in Section I, the result holds (even) for the inefficient but practically popular backtracking matching algorithm. We note that the converse direction of Theorem III.6 does not hold. That is, CDA is not a necessary condition for a rewb to be ReDoS invulnerable (even for the backtracking matching algorithm). For example, a^{**} over the alphabet $\{a\}$ is not CDA for the same reason that it is not CDA over the alphabet $\{a, b\}$ (cf. Example III.5), but it is not ReDoS vulnerable.

IV. Conversion to Invulnerable Regexes

In this section, we show a method to convert any nested DMFA to an equivalent ReDoS-invulnerable rewb. The correctness is proved by showing that the generated rewb is CDA and thus not ReDoS vulnerable by Theorem III.6.

Our method adopts the well-known state elimination method for converting an NFA into an equivalent regex (see, e.g., the textbook by Sipser [29]), and we in particular adopt the one proposed by Schmid [28] that extends the standard one to MFAs and rewbs. A subtle difference from Schmid's method is that whereas Schmid's method assumes that the input is a *normal-form* MFA which has no non-ϵ transitions that make memory status changes, we allow (D)MFAs that have such transitions. We extend Schmid's method this way because there are rewbs, such as $({}_1(a|b)^*)_1 c \backslash 1$ from Example II.3 that can be expressed as a nested DMFA as shown in Figure 2, but is seemingly impossible to express as a normal-form nested DMFA. Another subtle difference from standard state elimination methods is that we do the conversion per each final state by making copies of the automaton and then taking the union of the converted regexes, rather than doing the standard trick of adding a new final state to which the original final states have ϵ transitions. The reason for this difference is explained later in the section.

We now describe the conversion method. Our method adopts the approach of [28] that hierarchically converts an MFA to an equivalent rewb in the order of nesting of memory open-close pairs, from the inner-most ones to the outer-most ones. Let $A = (Q, \delta, q_0, F)$ be a nested MFA. For each memory index $i \in \mathbb{N}_{\geq 1}$, we define $\mathit{Ots}_A(i)$ and $\mathit{Cts}_A(i)$ to be respectively the

Algorithm 2 State elimination

Input: An eMFA $A = (Q, \delta, q_0, \{q_f\})$ with one final state q_f
Output: A rewb r satisfying $L(r) = L(A)$

1: Add to A a new initial state q_0' and the transition $q_0' \xrightarrow{\epsilon/\diamond} q_0$ (and make q_0 non-initial).
2: **while** a non-initial and non-final state remains in A **do**
3: Select a state q_{rip} that is neither initial nor final.
4: For every pair of states q_i and q_j of A such that $q_i \neq q_{rip}$, $q_j \neq q_{rip}$, $q_i \xrightarrow{r_1/s_1} q_{rip}$, $q_{rip} \xrightarrow{r_2/s_2} q_{rip}$, $q_{rip} \xrightarrow{r_3/s_3} q_j$, and $q_i \xrightarrow{r_4/s_4} q_j$, add the transition $q_i \xrightarrow{r/\diamond} q_j$ where $r = r_1 \cdot r_2^* \cdot r_3 | r_4$ to A.
5: Remove q_{rip} from A.
6: **return** $r_1 r_2^*$ where $q_0' \xrightarrow{r_1/\diamond} q_f$ and $q_f \xrightarrow{r_2/\diamond} q_f$

Algorithm 3 Open-close pair removal

Input: A nested MFA A
Output: A $\diamond$-only eMFA A' such that $L(A) = L(A')$

1: **while** Θ_A is not empty **do**
2: Pick $T = (i, tr, tr')$ such that $\Delta_A(T) = \emptyset$.
3: Make an eMFA A_T that is a copy of A except deleting all transitions in $Cts_A(i)$, letting q be the initial state and p' be the only final state where $tr = (p, \alpha, (_{\boldsymbol{i}}, q)$ and $tr' = (p', \alpha',)_{\boldsymbol{i}}, q')$.
4: Apply Algorithm 2 on A_T to obtain a rewb r.
5: Add the transition $(p, ({}_i \alpha r)_i \alpha', \diamond, q')$ to A.
6: Delete T from Θ_A and $\Delta_A(T')$ for every $T' \in \Theta_A$.
7: If tr is not in Θ_A then delete tr from A.
8: If tr' is not in Θ_A then delete tr' from A.
9: Replace each remaining transition $(q, \alpha, s, q') \in \delta$ where $s \neq \diamond$ with $(q, \alpha, \diamond, q')$.

set of $(_{\boldsymbol{i}}$ and $)_{\boldsymbol{i}}$ transitions. That is, $Ots_A(i) = \{(q, \alpha, s, q') \in \delta \mid s = (_{\boldsymbol{i}}\}$ and $Cts_A(i) = \{(q, \alpha, s, q') \in \delta \mid s =)_{\boldsymbol{i}}\}$ We define the *nesting relation* $\Theta_A = \bigcup_{i \in \mathbb{N}_{\geq 1}} \{i\} \times Ots_A(i) \times Cts_A(i)$. We define the binary relation $\lhd_A \subseteq \Theta_A \times \Theta_A$ as follows: $(i, tr_i, tr_i') \lhd_A (j, tr_j, tr_j')$ if there is a path that starts from tr_j and ends in tr_j' and takes transitions tr_j, tr_i, tr_i', and tr_j' in exactly this order such that no $)_{\boldsymbol{j}}$ transition is taken between tr_j and tr_j', and no $)_{\boldsymbol{i}}$ transition is taken between tr_i and tr_i'. By an argument similar to Lemma 15 of [28], we can show the following.

Lemma IV.1 ([28]). *If $(i, tr_i, tr_i') \lhd_A (j, tr_j, tr_j')$ then $i \neq j$. The relation $\lhd_A$ is irreflexive, transitive, and asymmetric.*

We write $\Delta_A(i, tr_1, tr_2)$ for the set $\{(j, tr_1', tr_2') \in \Theta_A \mid (j, tr_1', tr_2') \lhd_A (i, tr_1, tr_2)\}$. Algorithm 3 overviews the hierarchical conversion process that iteratively removes the open-close pairs starting from the inner-most ones. At each iteration, it uses Algorithm 2 to convert an automaton to an equivalent rewb.

Algorithm 2 is an adoption of the standard state elimination method for converting a given NFA to an equivalent regex (see, e.g., [29]), except that it is restricted to be given an automaton with only one final state. The restriction is satisfied when it is called by Algorithm 3, and as explained later, the top-level conversion algorithm (Algorithm 4) satisfies it by making a copy of the MFA per each final state. The restriction is needed because the standard trick of adding a new final state to which the original final states have ϵ transitions cannot be done. Namely, doing that would violate an invariant regarding a property called *strong disjointness* that we use to prove the correctness of our method (cf. the proof of Theorem IV.21).

Algorithm 4 Top-level conversion

Input: A nested DMFA A
Output: A rewb r such that $L(r) = L(A)$

1: Apply Algorithm 3 to remove non-$\diamond$ transitions from A.
2: $r \leftarrow \emptyset$
3: **for each** final state q_j in A **do**
4: Make a copy A_j of A with only q_j marked as final.
5: Apply Algorithm 2 on A_j to obtain a rewb r_j.
6: $r \leftarrow r | r_j$.
7: **return** r

As in the standard ones, our state elimination uses as intermediary *extended* automata that have expressions as transitions. We call a configuration *σ-initial* if it is of the form (q_0, σ) where q_0 is the initial state. We say that a path is σ-initial if it starts from the σ-initial state. For a rewb r and memories σ, let us write $Res_\sigma(r)$ for the set of pairs (w, σ') such that there exists a σ-initial run that matches w with σ' being the final memories, that is, $Res_\sigma(r) = \{(word(\pi), \sigma') \mid \pi \text{ is a } \sigma\text{-initial path of } MFA(r) \wedge last(\pi) = (q_f, \sigma')\}$ where q_f is the unique final state of $MFA(r)$. We write $L_\sigma(r)$ for $Res_\sigma(r)$ projected onto the first element, that is, $L_\sigma(r) = \{w \mid (w, \_) \in Res_\sigma(r)\}$.

Definition IV.2 (Extended MFA)**.** An *extended MFA* (eMFA) is an MFA with the transition function $\delta : Q \times Rewbs \times ScOps \to \mathcal{P}(Q)$, where $Rewbs$ is the set of rewbs. The transition $(q, \sigma) \xrightarrow{w,r,s} (q', \sigma')$ is defined to be the smallest relation satisfying: $\delta(q, r, s) \ni q'$ and $\sigma' = update(\sigma'', s, w)$ where $(w, \sigma'') \in Res_\sigma(r)$. We often call r the *label* of the transition.

Note that an ordinary MFA can be seen as an eMFA with the transition labels restricted to ϵ, a character, or a backreference. Also, for simplicity, we assume that for there is exactly one r/s labeling a transition between a pair of states. That is, for any $q, q' \in Q$, there is a unique r/s such that $q \xrightarrow{r/s} q'$. Note that any DMFA can be viewed as such an EMFA by taking the union of labels of the transitions between the states (letting the label be $\emptyset/\diamond$ when there is no transition from q to q'). The top-level conversion algorithm is shown in Algorithm 4. It first removes memory open and close operations by Algorithm 3. Then, as remarked before, it makes copies of the obtained eMFA per each final state to convert it to an equivalent regex by Algorithm 2, and returns the union of the obtained regexes. By an argument similar to Lemma 17 of [28], one can show

that the top-level conversion algorithm outputs a rewb that is equivalent to the input DMFA.

Theorem IV.3 ([28])**.** *Algorithm 4 returns a rewb that is equivalent to the input DMFA.*

The main technical contribution of this section is Theorem IV.21 which states that not only is the rewb returned by Algorithm 4 equivalent to the input automaton, but is also CDA. By Theorem III.6, this implies that the algorithm returns a ReDoS invulnerable rewb. The rest of the section is devoted to the proof of Theorem IV.21. We first prove a technical lemma concerning paths of (non-extended) MFAs.

Lemma IV.4. *Let π be a σ-initial path of an MFA A. Then, for any $u \preceq word(\pi)$, there is a σ-initial path of A that matches u.*

Proof. Either there is a prefix π' of π such that $word(\pi') = u$, or there is a prefix $\pi'' \cdot (q_1, \sigma_1) \xrightarrow{w} (\_, \_)$ of π where the last step takes a transition $(q_1, \backslash \boldsymbol{i}, \_, \_) \in \delta$ such that $w \preceq \sigma_1(i).con$ and $word(\pi'') \prec u \prec word(\pi'')w$. Here, $w = \sigma_1(i).con$ when case (2) of Definition II.2 is taken. In the former case, π' is a σ-initial path of $MFA(r)$ that matches u. In the latter case, the path $\pi'' \cdot (q_1, \sigma_1) \xrightarrow{w'} (\mathsf{stuck\text{-}st}, \sigma_1)$ where $word(\pi'')w' = u$ is a σ-initial path of $MFA(r)$ that matches u. Note that, in the latter case, case (3) of Definition II.2 is necessarily taken. □

Next, we introduce key notions for our proof called *strong disjointness* (SD) and *self strong disjointness* (SSD). Roughly, SD says that an accepting run in one regex has no paths, accepting or not, matching the same string in the other regex. Similarly, SSD says that an accepting run in a regex has no other paths, accepting or not, matching the same string in the same regex. To our knowledge, these notions are novel, for both pure regexes and rewbs. As we shall show next, they also exhibit interesting closure properties that turn out to be useful for the proof.

Definition IV.5 (SD)**.** We say that rewbs r_1 and r_2 are *strongly disjoint* (SD), written $r_1 \not\sim r_2$, if for all memories σ and $w \in L_\sigma(r_1)$, there is no σ-initial (accepting or non-accepting) path in $MFA(r_2)$ that matches w, and vice versa.

By definition, SD is symmetric, that is, $r_1 \not\sim r_2$ iff $r_2 \not\sim r_1$. Note that SD implies ordinary disjointness, that is, if $r_1 \not\sim r_2$ then $L_\sigma(r_1) \cap L_\sigma(r_2) = \emptyset$ for all σ. However, the converse does not generally hold: for example, a and ab are disjoint but not SD. Hence, the name "strong" disjointness. We show that SD is preserved by concatenations of the following form.

Lemma IV.6. *If $r_1 \not\sim r_2$, then $r_1 r \not\sim r_2$.*

Proof. Suppose for contradiction that there are $w \in L_\sigma(r_1 r)$ and a σ-initial path π of $MFA(r_2)$ such that $word(\pi) = w$. Then, there must be $u \preceq w$ such that $u \in L_\sigma(r_1)$. By Lemma IV.4, there must be a σ-initial path in $MFA(r_2)$ that matches u, which contradicts the assumption that $r_1 \not\sim r_2$. Now, suppose for contradiction that there are $w \in L_\sigma(r_2)$ and a σ-initial path π of $MFA(r_1 r)$ such that $word(\pi) = w$. If π ends in $MFA(r_1)$, then this contradicts $r_1 \not\sim r_2$. Therefore, there must be a prefix $u \preceq w$ such that $u \in L_\sigma(r_1)$. But by Lemma IV.4, there must be a σ-initial path in $MFA(r_2)$ that matches u, which contradicts $r_1 \not\sim r_2$. □

The following is immediate from the definition of SD.

Lemma IV.7. *If $r \not\sim r_1$ and $r \not\sim r_2$ then $r \not\sim r_1|r_2$.* □

Next, we formally define SSD. We say that a path is *maximal* if it does not end in a state that has exactly one outgoing transition, which additionally is labeled by ϵ.

Definition IV.8 (SSD)**.** We say that r is *self strongly disjoint* (SSD), if for any σ and any $w \in L_\sigma(r)$, there is exactly one maximal ϵcf σ-initial path of $MFA(r)$ that matches w, namely the ϵcf accepting run that witnesses $w \in L_\sigma(r)$.

We show the following property of SSD.

Lemma IV.9. *Suppose that r is SSD and $w \in L_\sigma(r)$. Then,*
(1) *for any $u \prec w$, $u \notin L_\sigma(r)$; and*
(2) *for any $u \succ w$, there is no σ-initial path, accepting or not, that matches u.*

Proof. For (1), let π be the maximal ϵcf σ-initial path of $MFA(r)$ that accepts w. Applying the construction in the proof of Lemma IV.4, we obtain from π a non-accepting maximal ϵcf σ-initial path π' that matches u. Then, by SSD of r, it must be the case that $u \notin L_\sigma(r)$. To show (2), for contradiction, suppose that there is a σ-initial path π that matches u. Then, since $w \prec u$, by applying the same construction, we obtain from π a non-accepting maximal ϵcf σ-initial path that matches w. But since $w \in L_\sigma(r)$, this contradicts the assumption that r is SSD. □

We show that SSD is closed under concatenation.

Lemma IV.10. *If r_1 and r_2 are SSD, then so is $r_1 r_2$.*

Proof. Suppose that $w \in L_\sigma(r_1 r_2)$, then there is $u \preceq w$ such that $u \in L_\sigma(r_1)$. By SSD of r_1 and Lemma IV.9 (1), such u is unique, and by Lemma IV.9 (2), if $u \prec w$ then there is no σ-initial path of $MFA(r_1)$, accepting or not, that matches w. Also, by SSD of r_1, there is exactly one maximal ϵcf σ-initial path π_1 of $MFA(r_1)$ that matches (and accepts) u. Let σ' be the final memories of that path. Then, by SSD of r_2, there is exactly one maximal ϵcf σ'-initial path π_2 of $MFA(r_2)$ that matches (and accepts) $u \backslash w$. Therefore, there is exactly one maximal ϵcf σ-initial path of $MFA(r_1 r_2)$ that matches (and accepts) w, namely, $\pi_1 \cdot (q_{f1}, \sigma') \xrightarrow{\epsilon} (q_{02}, \sigma') \cdot \pi_2$ where q_{f1} is the final state of $MFA(r_1)$ and q_{02} is the initial state of $MFA(r_2)$. □

By contrast, SSD is not closed under union in general, that is, $r_1|r_2$ may not be SSD even when r_1 and r_2 are (a simple counterexample is $r_1 = r_2 = a$). However, the following lemma shows that the union is SSD if $r_1 \not\sim r_2$.

Lemma IV.11. *If r_1 and r_2 are SSD and $r_1 \not\sim r_2$, then $r_1|r_2$ is SSD.*

Proof. Suppose $w \in L_\sigma(r_1|r_2)$. Either $w \in L_\sigma(r_1)$ or $w \in L_\sigma(r_2)$. Suppose that $w \in L_\sigma(r_1)$ (the case $w \in L_\sigma(r_2)$ is symmetric), and let π_1 be the unique maximal ϵcf σ-initial path of $MFA(r_1)$ that matches and accepts w. By $r_1 \not\sim r_2$, there is no path, accepting or not, that matches w in $MFA(r_2)$. Therefore, there is exactly one maximal ϵcf σ-initial path of $MFA(r_1|r_2)$ that matches (and accepts) w, namely, $(q_0, \sigma) \xrightarrow{\epsilon} (q_{01}, \sigma) \cdot \pi_1 \cdot (q_{f1}, \sigma') \xrightarrow{\epsilon} (q_f, \sigma')$ where q_0, q_f are the initial and the final states of $MFA(r_1|r_2)$, q_{01}, q_{f1} are the initial and the final states of $MFA(r_1)$, and σ' is the final memories of π_1. □

Likewise, SSD is not closed under Kleene star in general, that is, r^* may not be SSD even when r is (a simple counterexample is $r = a$). However, we show that SSD is preserved when the Kleene star is "guarded" by an SSD rewb that is strongly disjoint from r.

Lemma IV.12. *If r_1 and r_2 are SSD and $r_1 \not\sim r_2$, then $r_1^* r_2$ is SSD.*

Proof. Let π be a maximal ϵcf σ-initial accepting path of $MFA(r_1^* r_2)$, and $w = word(\pi)$. We may assume that $w \neq \epsilon$. It must be the case that π is one of the form:

$$(q_0, \sigma) \xrightarrow{\epsilon} (q_{fs}, \sigma) \xrightarrow{\epsilon} (q_{02}, \sigma) \cdot \pi_2 \cdot (q_{f2}, \_) \xrightarrow{\epsilon} (q_f, \_)$$

where q_{fs} is the final state of $MFA(r_1^*)$, q_{02} and q_{f2} are respectively the initial state and the final state of $MFA(r_2)$, and π_2 is a maximal ϵcf σ-initial accepting path of $MFA(r_2)$, or of the form:

$$\begin{aligned}(q_0, \sigma_0) \xrightarrow{\epsilon} (q_{01}, \sigma_0) \cdot \pi_{11} \cdot (q_{f1}, \sigma_1) \xrightarrow{\epsilon} (q_{01}, \sigma_1) \cdot \pi_{12} \cdots \\ \pi_{1n} \cdot (q_{f1}, \sigma_n) \xrightarrow{\epsilon} (q_{fs}, \sigma_n) \xrightarrow{\epsilon} (q_{02}, \sigma_n) \cdot \\ \pi_2 \cdot (q_{f2}, \_) \xrightarrow{\epsilon} (q_f, \_)\end{aligned}$$

for some $n \geq 1$ where $\sigma_0 = \sigma$, q_{01} and q_{f1} are respectively the initial and the final state of $MFA(r_1)$, π_{1i} is an ϵcf σ_{i-1}-initial accepting path of $MFA(r_1)$ for each $i \in \{1, \ldots, n\}$, and π_2 is a maximal ϵcf σ_n-initial accepting path of $MFA(r_2)$.

For the former case, by $r_1 \not\sim r_2$, there cannot be a σ-initial path in $MFA(r_1)$ matching $w = word(\pi_2)$, and by SSD of r_2, π_2 is the only σ-initial path in $MFA(r_2)$ matching w. Therefore, π is the only maximal ϵcf σ-initial path matching w.

For the latter case, let $w_{1i} = word(\pi_{1i})$ for each $i \in \{1, \ldots, n\}$, and $w_2 = word(\pi_2)$. Clearly, $w = w_{11} \ldots w_{1n} w_2$. By SSD of r_1, π_{1i} is the only ϵcf maximal σ_{i-1}-initial path of $MFA(r_1)$ matching w_{1i}, and by Lemma IV.9 (1), $u \notin L_{\sigma_{i-1}}(r_1)$ for any $u \prec w_{1i}$. Also by Lemma IV.9 (2), for any $u \succ w_{1i}$, there is no path in $MFA(r_1)$ that matches u. Also, by $r_1 \not\sim r_2$ and Lemma IV.4, no σ_{i-1}-initial path of $MFA(r_2)$ matches any $u \succeq w_{1i}$. Finally, by SSD of r_2, π_2 is the only maximal ϵcf σ_n-initial path of $MFA(r_2)$ matching w_2, and by $r_1 \not\sim r_2$ and Lemma IV.4, no σ_n-initial path of $MFA(r_1)$ matches w_2 and $u \notin L_{\sigma_n}(r_1)$ for any $u \preceq w_2$. Therefore, π is the only maximal ϵcf σ-initial path that matches w. □

We show that SD rewbs remain SD when prepended with an SSD rewb.

Lemma IV.13. *Let r_1 be SSD and $r_2 \not\sim r_2'$. Then, $r_1 r_2 \not\sim r_1 r_2'$.*

Proof. Let π be a σ-initial accepting path of $MFA(r_1 r_2)$, and $w = word(\pi)$. It suffices to show that there is no σ-initial path matching w in $MFA(r_1 r_2')$ (the other case is symmetric). For contradiction, let π' be a σ-initial path of $MFA(r_1 r_2')$ matching w. Let π_1 be the prefix of π within the $MFA(r_1)$ part, and let σ' be the final memories of π_1. By SSD of r_1 and Lemma IV.9, it must be the case that π_1 is also a prefix of π'. That is, $\pi' = \pi_1 \pi_2'$ for some σ'-initial path π_2' of $MFA(r_2')$. But then, π_2 where $\pi = \pi_1 \pi_2$ is a σ'-initial accepting path of $MFA(r_2)$ that matches $word(\pi_2')$, which contradicts $r_2 \not\sim r_2'$. □

The next lemma states that an analogous result holds when we prepend the SD rewbs by a Kleene star of an SSD rewb, provided that the latter rewb is also SD with the former two.

Lemma IV.14. *Let r_1 be SSD, $r_1 \not\sim r_2$, $r_1 \not\sim r_2'$, and $r_2 \not\sim r_2'$. Then, $r_1^* r_2 \not\sim r_1^* r_2'$.*

We extend the notion of degree of ambiguity from Section III to paths starting from σ-initial configurations. Formally, given an MFA A, the degree of ambiguity of A with respect to a string w and memories σ is defined by: $da_\sigma(A, w) = |\{\pi \in InitPaths_\sigma(A) \mid \pi \text{ is } \epsilon cf \wedge word(\pi) = w\}|$ where $InitPaths_\sigma(A)$ is the set of σ-initial paths of A. Likewise, we define the *σ-initial degree of ambiguity* of rewbs by: $da_\sigma(r, w) = da_\sigma(MFA(r), w)$. We define *all-memories degree of ambiguity* by: $ada(r, w) = \max_\sigma da_\sigma(r, w)$, and $ada(r) = \max_{w \in \Sigma^*} ada(r, w)$, letting $ada(r) = \infty$ if there is no $k \in \mathbb{N}$ such that $da_\sigma(r, w) \leq k$ for all $w \in \Sigma^*$ and memories σ. We say that a rewb r is *all-memories constant degree ambiguous* (ACDA) if $ada(r) \neq \infty$. Clearly from the definition, ACDA implies CDA. It is easy to see that ACDA is closed under union.

Lemma IV.15. *If r_1 and r_2 are ACDA, then so is $r_1|r_2$.* □

By contrast, ACDA is not closed under concatenation in general, that is, $r_1 r_2$ may not be ACDA even when r_1 and r_2 are. A counterexample is $r_1 = ({}_1(a|b)^*)_1$ and $r_2 = \backslash 1$ (cf. Example III.5). However, we can show that the concatenation is ACDA when the first rewb is also SSD.

Lemma IV.16. *If r_1 is SSD and ACDA and r_2 is ACDA, then $r_1 r_2$ is ACDA.*

ACDA is not closed under Kleene star in general, that is, r^* may not be ACDA even when r is (a simple counterexample is $r = a|a$). However, we can show the following result that is similar to Lemma IV.12.

Lemma IV.17. *If r_1 is SSD and ACDA, r_2 is ACDA, and $r_1 \not\sim r_2$, then $r_1^* r_2$ is ACDA.*

By an argument analogous to the above proof (i.e., let $r_2 = \emptyset$ in the above), we also obtain the following.

Corollary IV.18. *If r is SSD and ACDA, then r^* is ACDA.*

We show that the following invariants hold for the intermediate eMFAs constructed by Algorithm 2.

(I1) For any label r, r is SSD.
(I2) For any label r, r is ACDA.
(I3) For any r_1 and r_2 labeling two different transitions from a same state, $r_1 \not\sim r_2$.

Lemma IV.19. *Suppose Algorithm 2 is given an input eMFA that satisfies (I1)-(I3). Then, (I1)-(I3) hold for the eMFA A whenever the loop entry at line 2 is reached.*

Proof. We prove by induction on the number of times line 2 is reached. It is easy to see that (I1)-(I3) all hold for A when line 2 is reached for the first time.

Now, suppose as induction hypothesis that (I1)-(I3) hold for A at line 2. We show that each remains true the next time line 2 is reached. We first show (I1) and (I2) holds. Each newly introduced label is of the form $(r_1 r_2^* r_3)|r_4$, where r_1, r_2, r_3 and r_4 are labels on the transitions of A from the previous iteration such that r_1 and r_4 are labels on two different transitions from a same state and likewise for r_2 and r_3.

(I1) SSD of $r_2^* r_3$ follows from Lemma IV.12 since r_2 and r_3 are SSD and $r_2 \not\sim r_3$ by induction hypotheses (I1) and (I3). Then, SSD of $r_1 r_2^* r_3$ follows from Lemma IV.10 and induction hypothesis (I1) that r_1 is SSD. Finally, SSD of $(r_1 r_2^* r_3)|r_4$ follows from Lemma IV.7 and Lemma IV.6 since r_4 is SSD and $r_1 \not\sim r_4$ by induction hypotheses (I1) and (I3).
(I2) ACDA of $r_2^* r_3$ follows from Lemma IV.17 since r_2 and r_2 are ACDA, r_2 is SSD and $r_2 \not\sim r_3$ by induction hypotheses (I1), (I2), and (I3). Then, ACDA of $r_1 r_2^* r_3$ follows from Lemma IV.16 since r_1 is SSD and ACDA by induction hypotheses (I1) and (I2). Therefore, ACDA of $(r_1 r_2^* r_3)|r_4$ from Lemma IV.15 since r_4 is ACDA by induction hypothesis (I2).

Next, we show (I3). Let r and r' be labels on two different transitions from a same state. If both are transitions of A from the previous iteration, then (I3) holds immediately by induction hypothesis (I3). So, it suffices to consider the case where at least one of them is a label on a newly introduced transition which we assume, without loss of generality, is r. Let $r = (r_1 r_2^* r_3)|r_4$ where r_1, r_2, r_3, and r_4 are labels on the transition of A of A from the previous iteration such that r_1 and r_4 are labels on two different transitions from a same state and likewise for r_2 and r_3. If r' is a label on a transition of A from the previous iteration, then $r_1 r_2^* r_3 \not\sim r'$ follows from Lemma IV.6 since $r_1 \not\sim r'$ by induction hypothesis (I3). Therefore, $(r_1 r_2^* r_3)|r_4 \not\sim r'$ follows from Lemma IV.7 since $r_4 \not\sim r'$ by induction hypothesis (I3). Otherwise, r' is also a label of a newly introduced transition. It must be the case that $r' = (r_1 r_2^* r_3')|r_4'$ for some r_3' and r_4' such that r_2, r_3, and r_3' are mutually strongly disjoint and so are r_1, r_4, and r_4' by induction hypothesis (I3). Therefore, $r_2^* r_3 \not\sim r_2^* r_3'$ follows from Lemma IV.14 since r_2 is SSD by induction hypotheses (I1), and so $r_1 r_2^* r_3 \not\sim r_1 r_2^* r_3'$ follows from Lemma IV.13 since r_1 is SSD by induction hypothesis (I1). Therefore, $r_1 r_2^* r_3'$, $r_1 r_2^* r_3$, r_4, and r_4' are mutually strongly disjoint by Lemma IV.6, and so $(r_1 r_2^* r_3)|r_4 \not\sim (r_1 r_2^* r_3')|r_4'$ by Lemma IV.7. □

We prove a similar lemma for the intermediate eMFAs constructed during Algorithm 3

Lemma IV.20. *Suppose Algorithm 3 is given an input eMFA that satisfies (I1)-(I3). Then, (I1)-(I3) hold for the eMFA A whenever the loop entry at line 1 is reached.*

We are now ready to prove the main theorem of this section.

Theorem IV.21. *Algorithm 4 returns an ACDA rewb.*

Proof. Because the input MFA A is DMFA, it satisfies conditions (I1)-(I3) (note that the fact that A is DMFA is crucial for (I3) to hold).Therefore, by Lemma IV.20, each A_j created at line 4 also satisfies (I1)-(I3), and by Lemma IV.19, each r_j is of the form $r_{1j} r_{2j}^*$ for some r_{1j} and r_{2j} that are SSD and ACDA. By Corollary IV.18, r_{2j}^* is ACDA, and so r_j is ACDA by Lemma IV.16. Thus, the rewb r returned by the algorithm is ACDA by Lemma IV.15. □

We obtain the following as a corollary of Theorems III.6, IV.3 and IV.21.

Corollary IV.22. *Algorithm 4 returns a rewb that is equivalent to the input DMFA and is not ReDoS vulnerable.*

Recall that any pure regex can be converted to an equivalent DFA (e.g., by applying the textbook subset construction [29]). Thus, we obtain:

Corollary IV.23. *Any pure regex can be converted to an equivalent ReDoS invulnerable regex.*

Note that, as shown by Schmid [28], not every rewb can be represented as a DMFA, and so our construction method cannot be used to convert every rewb to an equivalent ReDoS-invulnerable one. For example, our method cannot convert $({}_1(a|b)^*)_1 \backslash 1$ to an equivalent ReDoS invulnerable regex as the regex cannot to be represented by a DMFA (cf. Example II.4). Indeed, we shall show in the next section that a conversion method that can turn any rewb to an equivalent ReDoS invulnerable form is unlikely to exist. However, since CDA is closed under union (cf. Lemma IV.15), it is easy to extend our method to a rewb expressible by a finite union of nested DMFAs. That is, given nested DMFAs $A_1, \ldots, A_n$, we run our algorithm to obtain equivalent CDA rewbs $r_1, \ldots, r_n$, and return $r_1 | \ldots | r_n$ which is also guaranteed to be CDA and therefore ReDoS invulnerable. For example, we can convert a rewb of the form $r|r'$ for $r = (({}_1(a|b)^*(a|b)(a|b)^*)_1(c|d)\backslash 1)^*$ and any pure regex r' to an equivalent ReDoS invulnerable one, because r can be represented as a nested DMFA by an approach analogous to Figure 2 and any pure regex can be represented by a DFA (and hence by a nested DMFA).

The complexity of the conversion algorithm is polynomial in the input nested DMFA. More precisely, we have the following.

Theorem IV.24. *Given a nested DMFA $A = (Q, \delta, q_0, F)$ with Θ_A as the nesting relation, Algorithm 4 runs in time $O((|\Theta_A| + |F|)|Q|^3)$.*

We note that the converted regex can be exponentially larger than the input DMFA in the worst case. This is expected because the conversion algorithm is guaranteed to return a pure regex when given a DFA, and it is well known that a pure regex must be exponentially larger than an equivalent DFA in general [17], [22].

V. Impossibility of Converting Rewbs in General

In this section, we show that, assuming a well-believed conjecture in parameterized complexity theory, there cannot be an algorithm (of any complexity) that converts an arbitrary rewb to an equivalent ReDoS invulnerable one. This is in stark contrast to the case of pure regexes every one of which can be converted to an equivalent ReDoS invulnerable regex by the algorithm given in the previous section (cf. Corollary IV.23). Recall that a parameterized problem is called *fixed-parameter tractable* (FPT) if it can be solved in time $f(k) \cdot poly(n)$ for some function f, where n is the input size and k is the parameter. W[1] is a class of parameterized problems that is believed to be not contained in FPT.[4] We next recall the following result from [18].

Theorem V.1 ([18])**.** *The matching problem for pattern languages [3] is W[1]-hard with respect to the input string length and with the size of the input pattern as the parameter.*

It is easy to see that rewbs can describe pattern languages (e.g., the pattern $axbx$ over the alphabet $\{a, b\}$ and a variable x can be described by the rewb $a(_1(a|b)^*)_1 b\backslash 1$). Therefore, the following corollary follows.

Corollary V.2. *Assuming that W[1] is not contained in FPT, there is no matching algorithm for rewbs that runs in time $f(r) \cdot poly(|w|)$ where w is the input string and r is the input rewb, for some function f.*

From this corollary, we immediately obtain the third main result of our paper:

Theorem V.3. *Suppose that $\mathcal{A}_m$ is a matching algorithm for rewbs. Assuming that W[1] is not contained in FPT, there is no algorithm that converts a given rewb r to an equivalent rewb r' such that $TIME_{\mathcal{A}_m}(r', w) \in O(|w|)$ where $TIME_{\mathcal{A}_m}(r', w)$ is the worst case running time of $\mathcal{A}_m$ on the input rewb r' and the input string w.*

Proof. Assume that W[1] is not contained in FPT. Suppose for contradiction that we have a conversion algorithm $\mathcal{A}_c$ that, given a rewb r, returns r' that is equivalent to r and $TIME_{\mathcal{A}_m}(r', w) \in O(|w|)$. This gives an FPT algorithm for matching rewbs as follows: this algorithm first converts the input rewb r by $\mathcal{A}_c$ and then runs $\mathcal{A}_m$ on the converted rewb and the input string w. It is easy to see that this algorithm indeed decides whether $w \in L(r)$ and is also FPT. Namely, it runs in time $TIME_{\mathcal{A}_c}(r) \cdot O(|w|)$ where $TIME_{\mathcal{A}_c}(r)$ is the worst-case running time of $\mathcal{A}_c$ on the input rewb r. However, by Corollary V.2, such an algorithm cannot exist, thus obtaining a contradiction. □

We note that the complexity of the conversion algorithm $\mathcal{A}_c$ does not matter to the argument above. Additionally, the statement of the theorem can be strengthened to state that there cannot be a method such that the converted rewb can be matched in time polynomial in the length of the input string for some fixed-degree polynomial. Also, by Theorem III.6, the above theorem implies that there cannot be a method that converts the given rewb to an equivalent CDA one.

As remarked in Section I, $\mathcal{A}_m$ can be any algorithm for matching rewbs. In particular, $\mathcal{A}_m$ can but not necessarily be the inefficient backtracking matching algorithm. Finally, we note that the result does not contradict the positive result from Section IV as it only rules out existence of methods that can convert *every* rewb to an equivalent ReDoS-invulnerable form (assuming W[1] $\neq$ FPT), and allows methods that can convert *some* rewbs to such a form.

VI. Related Work

There is a substantial amount of prior work on ReDoS and rewbs. In this section, we mention ones that are most closely related to the contributions of this paper.

Regarding the degree of ambiguity of regexes or automata, Weber and Seidl [35] have shown that CDA for an NFA can be characterized by a certain simple condition and also that the problem of determining if the given NFA is CDA is decidable in polynomial time. However, their work only considers NFAs without ϵ transitions, and while there have been proposals for using their characterization to detect ReDoS vulnerability for pure regexes, as pointed out by [11] (cf. the discussion at the end of Section III of their paper), such methods can be unsound and can classify an actually ReDoS vulnerable regex as invulnerable when the NFA representing the regex contains ϵ transitions.

In this paper, we have shown that, when ϵ transitions are properly considered, CDA indeed becomes a sufficient condition for guaranteeing ReDoS invulnerability for pure regexes. Additionally, we have extended the notion of the degree of ambiguity to MFAs and shown that CDA of those becomes a sufficient condition for ReDoS invulnerability for regexes with backreferences. That said, we do not have simple conditions in the style of [35] that characterize CDA or algorithms for deciding if the given regex or automaton is CDA, neither for the case of pure regexes or NFAs but with ϵ transitions, nor for the case further extended with backreferences. While we suspect that the former case is not too difficult and probably doable by a minor extension to the approach of [35],[5] the latter case seems to be a non-trivial open problem especially because even a basic problem such as universality is known to be undecidable for rewbs [19].

[4] We refer to a textbook [16] for more details on parameterized complexity.

[5] There is a prior work [2] that considered this problem, but their treatment of ϵ transitions is unsuitable for characterizing ReDoS.

Regarding methods to convert a given regex or automaton to a ReDoS invulnerable regex, Book et al. [8] have shown that applying state elimination to a DFA yields an "unambiguous" regex. However, they consider a weak form of unambiguity that is insufficient for guaranteeing ReDoS invulnerability. For example, a ReDoS vulnerable regex a^{**} over the alphabet $\{a, b\}$ would be considered unambiguous according to their definition. Van der Merwe et al. [34] have also proposed a conversion method for pure regexes that uses state elimination, but without a proof of correctness. To our knowledge, our work is the first to give a provably-correct state-elimination-based method to convert a regex or an automaton to an equivalent ReDoS invulnerable regex. In particular, we have proved that the output regex satisfies CDA and that CDA is a sufficient condition for ReDoS invulnerability. Additionally, while the prior methods mentioned above could only handle pure regexes, our conversion method supports regexes with backreferences.

A recent work by Chida and Terauchi [11] proposes a conversion method that can handle backreferences (as well as other popular extensions such as lookaheads). Their method takes as input a possibly-ReDoS-vulnerable regex r to be repaired, a finite set of *positive examples* $P \subseteq \Sigma^*$, and a finite set of *negative examples* $N \subseteq \Sigma^*$. It returns a regex r' satisfying (1) r' is *real-world-strongly-one-unambiguous* (RWS1U), (2) $P \subseteq L(r')$, (3) $N \cap L(r') = \emptyset$, and (4) r' has the shortest *edit distance* from r among the regexes satisfying (1)-(3). The edit distance between two regexes is the smallest number of AST nodes to remove from or add to the AST of one regex to convert it to that of the other. RWS1U is a sufficient condition for ReDoS invulnerability, and thus their method guarantees that the output regex r' is ReDoS invulnerable. However, besides that, as seen above, their method only guarantees that r' is consistent with the given examples and is syntactically close to the input regex in the sense of (4).

In particular, their method can output a regex that is semantically inequivalent to the input. As a concrete example, on the ReDoS-vulnerable regex $r = ({}_1(a|b)^{**})_1 c\backslash 1|aaaa$, the positive examples $\{aaaa, abcab\}$ and the negative examples $\{aa, abcba\}$, their method returns the regex $r' = ({}_1\emptyset^*)_1 c\backslash 1|a(a|b)(a|c)ab^*$.[6] While r' is indeed RWS1U (and therefore ReDoS invulnerable) and correctly classifies the given examples, it is clearly semantically inequivalent to r. In fact, our impossibility result (Theorem V.3) says that a semantics-preserving conversion method that works for every rewb and guarantees ReDoS invulnerability is unlikely to exist. That said, their work is in the context of *programming-by-examples* and assumes that a user of their method may want to repair not only ReDoS vulnerability but also possibly the semantics of the regex. Therefore, the fact that semantics is not preserved by their method may be considered a feature rather than a defect in the context of their work.

Finally, there has been much work on ReDoS-free non-backtracking matching algorithms such as Google's RE2 and the ones proposed in very recent papers [25], [21], [4]. However, as mentioned in Section I, these algorithms lack the support for backreferences. We think that our work sheds light on how to provide some support for backreferences while still avoiding ReDoS.

[6]The tool is available at https://github.com/NariyoshiChida/SP2022.

VII. Conclusion

In this paper, we have conducted a formal study of DoS vulnerability of regexes with and without backreferences. We have made the following three key contributions. First, we have extended the notion of the degree of ambiguity to regexes with backreferences and shown that constant degree of ambiguity is a sufficient condition for guaranteeing ReDoS invulnerability, for both the cases with and without backreferences. Secondly, we have proposed a method to convert a given finite union of deterministic memory automata to an equivalent CDA (and hence ReDoS invulnerable, by the first contribution) regex. This in particular has given a method that converts an arbitrary pure regex to an equivalent ReDoS invulnerable one. Finally, we have shown that, unlike for the case of pure regexes, assuming the well-believed conjecture that W[1] is not contained in FPT, there cannot be a method that converts an arbitrary regex with backreferences to an equivalent ReDoS invulnerable one. As remarked before, the first two positive ones hold even for the inefficient backtracking matching algorithm, whereas the third negative result holds for any matching algorithm.

As also discussed in Section VI, a possible direction for future work is to look for a simple condition that characterizes CDA for rewbs or MFAs in the style of Weber and Seidl's characterization of CDA for ϵ-transition-free NFAs [35], and investigate the decidability and the computational complexity of the problem of determining if the given rewb or MFA is CDA. Another possible direction for future work is to extend the study to regexes containing both backreferences and lookarounds. Like backreference, lookaround is a practically popular extension supported by most major regex engines, and recent papers [10], [33] have shown that regexes extended with both backreferences and lookarounds are significantly expressive, in particular, strictly more expressive than regexes extended with only backreferences or only lookarounds. While the conversion method proposed by Chida and Terauchi [11] can handle a regex containing both backreferences and lookarounds, as remarked in Section VI, their method may not preserve the semantics of the regex. How to ensure ReDoS freedom in the presence of both of these features remains an important but largely open problem.

Acknowledgment

We thank anonymous reviewers for their useful comments. We thank Timos Antonopoulos and Nariyoshi Chida for collaborating with us at early stages of this work. We especially owe Timos for the idea (and the name) of self strong disjointness. This work was supported by JSPS KAKENHI Grant Numbers JP23K24826, JP20K20625, and JP23K20380.

References

[1] Weidman Adar. Regular expression denial of service - ReDoS, 2017. URL: https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[2] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata. In *Developments in Language Theory, 12th International Conference, DLT 2008, Kyoto, Japan, September 16-19, 2008. Proceedings*, volume 5257 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2008. doi:10.1007/978-3-540-85780-8_8.

[3] Dana Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, 1980. doi:10.1016/0022-0000(80)90041-0.

[4] Aurèle Barrière and Clément Pit-Claudel. Linear matching of JavaScript regular expressions. *Proc. ACM Program. Lang.*, 8(PLDI):1336–1360, 2024. doi:10.1145/3656431.

[5] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, May 2016. doi:10.1109/TKDE.2016.2515587.

[6] Martin Berglund and Brink van der Merwe. Re-examining regular expressions with backreferences. *Theor. Comput. Sci.*, 940(Part):66–80, 2023. doi:10.1016/j.tcs.2022.10.041.

[7] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, 1973. doi:10.1016/S0019-9958(73)90851-6.

[8] Ronald Vernon Book, Shimon Even, Sheila A. Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Trans. Computers*, 20(2):149–153, 1971. doi:10.1109/T-C.1971.223204.

[9] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(6):1007–1018, 2003. doi:10.1142/S012905410300214X.

[10] Nariyoshi Chida and Tachio Terauchi. On lookaheads in regular expressions with backreferences. In *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.FSCD.2022.15, doi:10.4230/LIPICS.FSCD.2022.15.

[11] Nariyoshi Chida and Tachio Terauchi. Repairing DoS vulnerability of real-world regexes. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2060–2077. IEEE, 2022. doi:10.1109/SP46214.2022.9833597.

[12] Nariyoshi Chida and Tachio Terauchi. Repairing regular expressions for extraction. *Proc. ACM Program. Lang.*, 7(PLDI):1633–1656, 2023. doi:10.1145/3591287.

[13] Nariyoshi Chida and Tachio Terauchi. Repairing regex-dependent string functions. In Vladimir Filkov, Baishakhi Ray, and Minghui Zhou, editors, *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 294–305. ACM, 2024. doi:10.1145/3691620.3695005.

[14] J. C. Davis, F. Servant, and D. Lee. Using selective memoization to defeat regular expression denial of service (ReDoS). In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 543–559, Los Alamitos, CA, USA, may 2021. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00032, doi:10.1109/SP40001.2021.00032.

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

[16] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999. doi:10.1007/978-1-4612-0515-9.

[17] Andrzej Ehrenfeucht and H. Paul Zeiger. Complexity measures for regular expressions. *Journal of Compututer and System Sciences*, 12(2):134–146, 1976. doi:10.1016/S0022-0000(76)80034-7.

[18] Henning Fernau and Markus L. Schmid. Pattern matching with variables: A multivariate complexity analysis. *Information and Computation*, 242:287–305, 2015. URL: https://doi.org/10.1016/j.ic.2015.03.006, doi:10.1016/J.IC.2015.03.006.

[19] Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Syst.*, 53(2):159–193, 2013. URL: https://doi.org/10.1007/s00224-012-9389-0, doi:10.1007/S00224-012-9389-0.

[20] Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.*, 105:1–39, 2019. doi:10.1016/j.jcss.2019.04.001.

[21] Hiroya Fujinami and Ichiro Hasuo. Efficient matching with memoization for regexes with look-around and atomic grouping. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024*, volume 14577 of *Lecture Notes in Computer Science*, pages 90–118. Springer, 2024. doi:10.1007/978-3-031-57267-8_4.

[22] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008*, volume 5126 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2008. doi:10.1007/978-3-540-70583-3_4.

[23] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 1, USA, 2011. USENIX Association.

[24] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 21–30, Honolulu, Hawaii, October 2008. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/D08-1003.

[25] Konstantinos Mamouras and Agnishom Chattopadhyay. Efficient matching of regular expressions with lookaround assertions. *Proc. ACM Program. Lang.*, 8(POPL):2761–2791, 2024. doi:10.1145/3632934.

[26] Taisei Nogami and Tachio Terauchi. On the expressive power of regular expressions with backreferences. In *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*, volume 272 of *LIPIcs*, pages 71:1–71:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.MFCS.2023.71, doi:10.4230/LIPICS.MFCS.2023.71.

[27] Francesco Parolini and Antoine Miné. Sound static analysis of regular expressions for vulnerabilities to denial of service attacks. *Science of Compututer Programming*, 229:102960, 2023. URL: https://doi.org/10.1016/j.scico.2023.102960, doi:10.1016/J.SCICO.2023.102960.

[28] Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Information and Computation*, 249:1–17, 2016. doi:10.1016/j.ic.2016.02.003.

[29] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[30] Henry Spencer. A regular-expression matcher. In *Software solutions in C*, pages 35–71. 1994.

[31] Satoshi Sugiyama and Yasuhiko Minamide. Checking time linearity of regular expression matching based on backtracking. *Information and Media Technologies*, 9(3):222–232, 2014. doi:10.11185/imt.9.222.

[32] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. doi:10.1145/363347.363387.

[33] Yuya Uezato. Regular expressions with backreferences and lookaheads capture NLOG. In *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024, July 8-12, 2024, Tallinn, Estonia*, volume 297 of *LIPIcs*, pages 155:1–155:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.ICALP.2024.155, doi:10.4230/LIPICS.ICALP.2024.155.

[34] Brink van der Merwe, Nicolaas Weideman, and Martin Berglund. Turning evil regexes harmless. In *South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017*, pages 38:1–38:10. ACM, 2017. doi:10.1145/3129416.3129440.

[35] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, 88(2):325–349, 1991. doi:10.1016/0304-3975(91)90381-B.

[36] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016*, volume 9705 of *Lecture Notes in Computer Science*, pages 322–334. Springer, 2016. doi:10.1007/978-3-319-40946-7_27.

[37] Fang Yu, Ching-Yuan Shueh, Chun-Han Lin, Yu-Fang Chen, Bow-Yaw Wang, and Tevfik Bultan. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 189–200, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2931037.2931050.

APPENDIX
OMITTED PROOFS

Lemma IV.14. *Let r_1 be SSD, $r_1 \not\sim r_2$, $r_1 \not\sim r_2'$, and $r_2 \not\sim r_2'$. Then, $r_1^* r_2 \not\sim r_1^* r_2'$.*

Proof. Let π be an ϵcf σ-initial accepting path of $MFA(r_1^* r_2)$, and $w = word(\pi)$ (the case $w \in L_\sigma(r_1^* r_2')$ is symmetric). Without loss of generality, we assume that $w \neq \epsilon$. It must be the case that π is of the form:

$$(q_0, \sigma) \xrightarrow{\epsilon} (q_{fs}, \sigma) \xrightarrow{\epsilon} (q_{02}, \sigma) \cdot \pi_2 \cdot (q_{f2}, \_) \xrightarrow{\epsilon} (q_f, \_)$$

where q_{fs} is the final state of $MFA(r_1^*)$, q_{02} and q_{f2} are respectively the initial state and the final state of $MFA(r_2)$, and π_2 is an ϵcf σ-initial accepting path of $MFA(r_2)$, or of the form:

$$\begin{aligned}(q_0, \sigma_0) \xrightarrow{\epsilon} (q_{01}, \sigma_0) \cdot \pi_{11} \cdot (q_{f1}, \sigma_1) \xrightarrow{\epsilon} (q_{01}, \sigma_1) \cdot \pi_{12} \cdots \\ \pi_{1n} \cdot (q_{f1}, \sigma_n) \xrightarrow{\epsilon} (q_{fs}, \sigma_n) \xrightarrow{\epsilon} (q_{02}, \sigma_n) \cdot \\ \pi_2 \cdot (q_{f2}, \_) \xrightarrow{\epsilon} (q_f, \_)\end{aligned}$$

for some $n \geq 1$ where $\sigma_0 = \sigma$, q_{01} and q_{f1} are respectively the initial and the final state of $MFA(r_1)$, π_{1i} is an ϵcf σ_{i-1}-initial accepting path of $MFA(r_1)$ for each $i \in \{1, \ldots, n\}$, and π_2 is an ϵcf σ_n-initial accepting path of $MFA(r_2)$.

For the former case, suppose for contradiction that there is a σ-initial path of $MFA(r_1^* r_2')$ that matches w. Because $r_1 \not\sim r_2'$, there is no σ-initial path of $MFA(r_1)$ that matches any prefixes of w (including w itself). Therefore, it must be the case that there is a σ-initial path of $MFA(r_2')$ that matches w, which contradicts $r_2 \not\sim r_2'$.

For the latter case, suppose for contradiction that there is a σ-initial path π' of $MFA(r_1^* r_2')$ such that $word(\pi') = w$. Let $\pi = \pi_1 \pi_2$. By SSD of r_1 and Lemmas IV.9 and IV.4, it must be the case that $\pi' = \pi_1 \pi_2'$ for some σ_n-initial path π_2' of $MFA(r_2)$ (cf. the second case in the proof of Lemma IV.12). But $word(\pi_2) = word(\pi_2')$, and because $r_2 \not\sim r_2'$, such π_2' cannot exist, contradiction. □

Lemma IV.16. *If r_1 is SSD and ACDA and r_2 is ACDA, then $r_1 r_2$ is ACDA.*

Proof. Let π be an ϵcf σ-initial path in $MFA(r_1 r_2)$. Suppose that π ends in $MFA(r_1)$. Let $word(\pi) = w$. By SSD of r_1 and Lemma IV.9 (2), $u \notin L_\sigma(r_1)$ for any $u \prec w$, and so $da_\sigma(r_1 r_2, w) \leq da_\sigma(r_1, w) + da_{\sigma'}(r_2, \epsilon) + 1$ where $+da_{\sigma'}(r_2, \epsilon) + 1$ accounts for the case that w is accepted by r_1 and σ' is the final memories of the (necessarily unique as r_1 is SSD) σ-initial path of $MFA(r_1)$ that accepts w.

Otherwise, π ends in $MFA(r_2)$, and we can let $\pi = \pi_1 \cdot (q_{f1}, \sigma') \xrightarrow{\epsilon} (q_{02}, \sigma') \cdot \pi_2$ where π_1 is an ϵcf σ-initial accepting path of $MFA(r_1)$, π_2 is an ϵcf σ'-initial (possibly non-accepting) path of $MFA(r_2)$, q_{f1} is the final state of $MFA(r_1)$, and q_{02} is the initial state of $MFA(r_2)$. Let $w_1 = word(\pi_1)$ and $w_2 = word(\pi_2)$. By SSD of r_1 and Lemma IV.9 (1), π_1 is the unique ϵcf path of $MFA(r_1)$ that matches w_1 and $u \notin L(\pi_1)$ for any $u \prec w_1$ and by Lemma IV.9 (2), there is no σ-initial path of $MFA(r_1)$ that matches any $u \succ w_1$. Therefore, $da_\sigma(r_1 r_2, w) \leq da_{\sigma'}(r_2, w_2) + 1$ where $+1$ accounts for the case that $w_2 = \epsilon$. Thus, $r_1 r_2$ is ACDA. □

Lemma IV.17. *If r_1 is SSD and ACDA, r_2 is ACDA, and $r_1 \not\sim r_2$, then $r_1^* r_2$ is ACDA.*

Proof. Let π be an ϵcf σ-initial path of $MFA(r_1^* r_2)$, and $w = word(\pi)$. As in the proof of Lemma IV.14, without loss of generality, we assume that $w \neq \epsilon$.[7] It must be the case that π is of the form:

$$(q_0, \sigma) \xrightarrow{\epsilon} (q_{fs}, \sigma) \xrightarrow{\epsilon} (q_{02}, \sigma) \cdot \pi_2$$

where q_{fs} is the final state of $MFA(r_1^*)$, q_{02} is the initial stateof $MFA(r_2)$, and π_2 is an ϵcf σ-initial path of $MFA(r_2)$, or of the forms:

$$\begin{aligned}(q_0, \sigma_0) \xrightarrow{\epsilon} (q_{01}, \sigma_0) \cdot \pi_{11} \cdot (q_{f1}, \sigma_1) \xrightarrow{\epsilon} (q_{01}, \sigma_1) \cdot \pi_{12} \cdots \\ \pi_{1n} \cdot (q_{f1}, \sigma_n) \xrightarrow{\epsilon} (q_{fs}, \sigma_n) \xrightarrow{\epsilon} (q_{02}, \sigma_n) \cdot \pi_2\end{aligned}$$

or

$$(q_0, \sigma_0) \xrightarrow{\epsilon} (q_{01}, \sigma_0) \cdot \pi_{11} \cdot (q_{f1}, \sigma_1) \xrightarrow{\epsilon} (q_{01}, \sigma_1) \cdot \pi_{12} \cdots \pi_{1n}$$

possibly followed by ϵ transitions, for some $n \geq 1$ where $\sigma_0 = \sigma$, q_{01} and q_{f1} are respectively the initial and the final state of $MFA(r_1)$, π_{1i} is an ϵcf σ_{i-1}-initial path of $MFA(r_1)$ that is accepting except for possibly the last π_{1n} for the second case for each $i \in \{1, \ldots, n\}$, and π_2 is an ϵcf σ_n-initial path of $MFA(r_2)$.

In the first case, by $r_1 \not\sim r_2$ and Lemma IV.4, it must be the case that $u \notin L_\sigma(r_1)$ for any $u \preceq w$. Therefore, $da_\sigma(r_1^* r_2, w) \leq da_\sigma(r_2, w)$.

In the remaining two cases, by adopting an argument similar to Lemma IV.12, we can show that the part of the path up to the last accepting (for $MFA(r_1)$) π_{1j} is uniquely determined, and show that $ada(r_1^* r_2, w) \leq ada(r_1) + 2$ by an argument similar to Lemma IV.16 and the above, where $+2$ accounts for the second case with π_{1n} accepting as we can extend the path with two additional ϵ transitions, namely, to q_{fs} and subsequently to q_{02}. Thus, $r_1^* r_2$ is ACDA.

□

Lemma IV.20. *Suppose Algorithm 3 is given an input eMFA that satisfies (I1)-(I3). Then, (I1)-(I3) hold for the eMFA A whenever the loop entry at line 1 is reached.*

[7] Note that ruling out finitely many strings is fine because the degree of ambiguity for such strings is finitely bounded.

Proof. We prove by induction on the number of times line 1 is reached. The fact that (I1)-(I3) all hold for A when line 1 is reached for the first time is immediate from the assumption.

Now, suppose as induction hypothesis that (I1)-(I3) hold for A at line 1. We show that each remains true the next time line 1 is reached. Let $(p, ({}_i\alpha r)_i\alpha', \diamond, q')$ be the newly added transition. By inspection of Algorithm 2 and Lemma IV.19, $r = r_1 r_2^*$ for some r_1 and r_2 such that r_1 and r_2 are SSD and ACDA, and $r_2 \not\sim \alpha'$.

(I1) It suffices to show that the label on the new transition is SSD. By Lemma IV.12, $r_2^*\alpha'$ is SSD since α' is SSD by induction hypothesis (I1). Therefore, by Lemma IV.10, $\alpha r \alpha' = \alpha r_1 r_2^* \alpha'$ is SSD since α is SSD by induction hypothesis (I1). Thus, $({}_i\alpha r)_i\alpha'$ is SSD.

(I2) It suffices to show that the label on the new transition is ACDA. By Lemma IV.17, $r_2^*\alpha'$ since α' is ACDA by induction hypothesis (I2). Therefore, by Lemma IV.16, $\alpha r \alpha' = \alpha r_1 r_2^* \alpha'$ is ACDA since α is SSD and ACDA by induction hypotheses (I1) and (I2). Thus, $({}_i\alpha r)_i\alpha'$ is ACDA.

(I3) Suppose that $(p, r', \_)$ is a transition in A from the previous iteration. It suffices to show that $r' \not\sim ({}_i\alpha r)_i\alpha'$. By Lemma IV.6, $r' \not\sim \alpha r \alpha'$ since $r' \not\sim \alpha$ by induction hypothesis (I3). Thus, $r' \not\sim ({}_i\alpha r)_i\alpha'$.

□

Theorem IV.24. *Given a nested DMFA $A = (Q, \delta, q_0, F)$ with Θ_A as the nesting relation, Algorithm 4 runs in time $O((|\Theta_A| + |F|)|Q|^3)$.*

Proof. Algorithm 2 is a straightforward adaptation of the standard state elimination algorithm which runs, in the worst case, in time cubic in the number of states of the input eMFA. Algorithm 3 calls Algorithm 2 $|\Theta_A|$ times and, for each such call, do additional work which takes at most $O(|Q|^2)$ time. Therefore, Algorithm 3 runs in time $O(|\Theta_A|(|Q|^3))$. Finally, Algorithm 4 first calls Algorithm 3 and then runs Algorithm 2 at most $|F|$ many times. Therefore, the worst-case time complexity of Algorithm 4 is $O((|\Theta_A| + |F|)|Q|^3)$. □