Static Analysis of Regular Expressions

by

Nicolaas Hendrik Weideman

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science (Computer Science) in the Faculty of Science at Stellenbosch University

Supervisor: Prof. A. B. van der Merwe

December 2017

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2017

Abstract

Static Analysis of Regular Expressions

N. H. Weideman

Department of Computer Science,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MSc (Computer Science)

December 2017

Regular expressions are widely used throughout the programming community. In most cases, regular expressions allow for pattern matching tasks to be performed efficiently, but in some instances regular expression matching can be extremely slow. The exploit of the potential slowness of regular expression matching, is known as a regular expression denial of service attack. We investigate regular expression denial of service attacks, by approaching it from a computational complexity and automata theoretic point of view. A method for accurately modeling the matching time behaviour of a backtracking regular expression matcher, by using automata theoretic methods, is presented. We analyze our models by using the concept of ambiguity in nondeterministic finite-state automata. Our approach is evaluated on repositories of regular expressions often used in practice. Techniques for mitigating the vulnerability of backtracking regular expression matchers are investigated as a means to thwart regular expression denial of service attacks.

Uittreksel

Static Analysis of Regular Expressions

N. H. Weideman

Department of Computer Science,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Tesis: MSc (Computer Science)

Desember 2017

Reguliere uitdrukkings word gereeld gebruik in die skryf van sagteware. In die meeste gevalle stel sulke uitdrukkings mens in staat om patroonherkenningsprobleme op 'n doeltreffende manier op te los. Daar is egter sommige situasies waar hierdie proses uiters tydrowend kan wees. Die uitbuiting van sulke kwesbaarhede staan as 'n diensontseggingaanval bekend. Ons ondersoek hierdie aanvalle vanuit die oogpunt van berekeningskompleksiteit en outomateteorie. 'n Metode word gegee om die herkenningstyd van 'n terugspoor herkenner van reguliere uitdrukkings akkuraat te modelleer. Ons analiseer die modelle deur gebruik te maak van die konsep van dubbelsinnigheid in nie-deterministiese eindigetoestand-outomate. Die metodes word getoets deur dit toe te pas op magasyne van reguliere uitdrukkings wat in die praktyk gebruik word. Tegnieke om die kwesbaarheid van terugspoor herkenners van reguliere uitdrukkings te verbeter word ondersoek, met die doelwit om diensontseggingaanvalle te voorkom.

Acknowledgements

I wish to extend my sincerest gratitude to the following people and organisations for their support in writing this thesis. Firstly, I would like to thank my family for encouraging and supporting me throughout the entire process. Secondly, I would like to thank my supervisor, Professor A. B. van der Merwe for his guidance and sharing his experience and knowledge. Lastly, I would like to thank the Center for AI Research for financial support throughout my masters program.

Dedications

I would like to dedicate this thesis to my family, without whose never-ending support its completion would have been impossible.

Contents

List of Figures

List of Tables

Nomenclature

Notations

$\mathbb{N}$	The non-negative integers, i.e. $\{0, 1, 2, \ldots\}$.
$\mathbb{N}_+$	The positive integers, i.e. $\{1, 2, \ldots\}$.
Σ	A non-empty finite alphabet.
ε	The empty string.
Σ^ε	$\Sigma \cup \{\varepsilon\}$
$\lvert w \rvert$	The number of symbols in the sequence w.
$\lvert w \rvert_a$	The number of occurrences of the symbol a in the sequence w.
$\mathcal{P}(S)$	The powerset of set S.

Acronyms and Abbreviations

DFA	Deterministic Finite-state Automaton
NFA	Nondeterministic Finite-state Automaton
pNFA	prioritised Nondeterministic Finite-state Automaton

Chapter 1

Introduction

Regular expressions, which we refer to as *regexes* when discussing extended regular expressions as found in software libraries, provide a convenient way to describe the class of regular languages and are thus frequently studied in the field of formal language theory [1; 2; 3]. In addition, regexes provide a method for specifying patterns of interest in text in a compact way. This leads to regexes being used widely in programming.

It is assumed throughout most of the programming community that regex matching is fast. Even though this assumption is accurate in most cases, under some conditions regex matching can be extremely slow. These conditions can cause a regex to be vulnerable to a class of algorithmic complexity attacks, known as *regular expression denial of service (ReDoS)* attacks. In ReDoS, a program halts normal execution, due to it being stuck trying to determine if a particular string contains some pattern of interest, by using regex matching. A regex that is vulnerable to ReDoS is said to be *vulnerable* or *evil*.

There have been numerous accounts of ReDoS documented, some of which are listed in [4]. One account in particular is the outage of the popular question-and-answer website, Stack Overflow — a regex used to remove excessive whitespace from posts, turned out to be evil. This resulted in the website becoming unresponsive after a post with a large number of whitespace characters was submitted [5].

ReDoS, as far as we know, only happens when using a type of matcher known as a backtracking matcher. This thesis will focus almost exclusively on backtracking matchers. ReDoS is caused by the search algorithm used to determine if and how an input string is matched by a regex. For evil regexes, it is usually the case that a particular subexpression of the regex can match some prefix of the input string in multiple ways, but that the entire input string cannot be matched by the regex. The search algorithm then has to try and match the input string with the subexpression, rejecting and backtracking each time until all the multiple ways of matching the prefix with the subexpression has been attempted. The backtracking performed by a matcher for an evil regex is known as *catastrophic backtracking* [6].

Even though some regex matchers do not perform matching using backtracking, such as those used in the AWK programming language [7] and the regex engine of MySQL [8], these alternative matchers do not support various features found in typical backtracking matchers. One of these features (known as capturing) makes it possible to determine which substring was matched by a given subexpression of the regex. Note that the regex engine RE2 [9] is not susceptible to ReDoS and supports capturing, but again has other limitations in terms of supported features. Therefore, in spite of being susceptible to ReDoS, backtracking matchers are used in most popular programming languages such as Java and Python.

The goal of this thesis is to develop static analysis techniques that can be used to determine if a given regex is susceptible to ReDoS in the Java programming language. Java was chosen specifically, because it implements a backtracking matcher in its standard regex matching library. The backtracking matcher behaves similarly to many other regex libraries utilising the same type of matcher, such as the one used by the commandline tool Perl Compatible Regular Expressions [10]. However, unlike many other regex libraries that also use backtracking matchers, the one found in Java does not seem to implement arbitrary checks in the attempt to prevent catastrophic backtracking. Therefore, the matching time of a regex used with the Java matcher, should provide an upper bound for other matchers as well.

1.1 Thesis Outline

Our investigation into vulnerable regexes begins with a study of the fundamentals of both formal language theory and regexes in Chapters 2 and 3. Using this as a basis, Chapter 4 explains how regex matching works. In Chapter 5, the context of ReDoS in the field of information security is given, as well as other research done in this area. We begin the formulation of our approach by explaining the concept of ambiguity and how it is related to regexes in Chapter 6. Using the concept of ambiguity, we can decide when a regex is vulnerable, which is explained in Chapter 7, and described in more detail in Chapter 8. When a regex is classified as being vulnerable, it is required that the matcher shows nonlinear polynomial, or exponential matching time for some sequence of input strings of increasing length. Chapter 9 explains how such strings can be constructed. Finally, experimental results of running our analysis on repositories of commonly used regexes is given (Chapter 10), followed by an investigation into how vulnerable regexes can (in certain cases) be ameliorated (Chapter 11).

Chapter 2

Fundamentals of Formal Language Theory

2.1 Introduction

To understand regular expressions and what makes them vulnerable, we first need to define them formally. Since regular expressions are studied in formal language theory, it seems logical that we start by defining some relevant concepts therein. These concepts are discussed in more detail in the textbook *Introduction to the Theory of Computation* by Michael Sipser [11].

2.2 Alphabets, Strings and Languages

An *alphabet* is a nonempty finite set of symbols, while a *string* is a finite (perhaps empty) sequence of symbols from an alphabet. The *length* of a string w is the number of symbols in w, and is denoted by $|w|$. By $|w|_a$, we denote the number of symbols in the sequence w equal to the symbol a. Also, by w^k we denote k successive copies of the string w. If all symbols of some string w are from some alphabet Σ, we say that w is a string over Σ. When $|w| = 0$, we say that w is the empty string and denote it by ε. We assume that $\varepsilon \notin \Sigma$ and denote $\Sigma \cup \{\varepsilon\}$ by Σ^ε. Finally, a language is a subset (possibly empty) of the set of all strings over some alphabet.

2.3 Finite-state Automata

A deterministic finite-state automaton (DFA) is a state machine that reads input strings and either *accepts* or *rejects* such strings after the final symbol has been read. A DFA M, over an alphabet Σ, defines a language consisting of all input strings accepted by M. The set of input strings accepted by a DFA

is said to be the language *recognised* by the DFA, and is denoted as $\mathcal{L}(M)$. We formally define DFAs in Definition 2.1.

Definition 2.1. A DFA is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, such that:

1. Q is a finite nonempty set of states,
2. Σ is the finite input alphabet,
3. δ is the deterministic transition function where $\delta : Q \times \Sigma \to Q$,
4. $q_0 \in Q$ is the initial state, and
5. $F \subseteq Q$ is the set of accept states.

Let $|A|_Q = |Q|$ be the number of states and $|A|_\delta = |\Sigma||Q|$ the number of transitions in A.

Nondeterministic finite-state automata (NFA) are a generalisation of DFAs. As with DFAs, an NFA M, over an alphabet Σ, defines a language consisting of all the input strings accepted by M and the set of input strings accepted by an NFA is said to be the language *recognised* by the NFA, and is denoted as $\mathcal{L}(M)$.

While processing input strings symbol by symbol, the next state that an NFA will transition to is selected nondeterministically from a set of states determined by the current state and the next input symbol, or only by the current state if the NFA nondeterministically selects not to read an input symbol. Note that when no input symbol is consumed, we sometimes say that an ε is consumed, in the sense that the empty string is consumed. These transitions are referred to as *empty transitions* (or ε-transitions) and are labeled with the symbol ε.

Intuitively, nondeterminism can be thought of as if the NFA concurrently traverses all valid path options, while reading some input string. After the entire input string has been consumed, the machine accepts the input string if the input string caused at least one of these paths to end in a so-called *accept state*. It rejects the input string otherwise.

Let us now formally define an NFA.

Definition 2.2. An NFA is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, such that:

1. Q is a finite nonempty set of states,
2. Σ is the finite input alphabet,
3. δ is the transition function where $\delta : Q \times \Sigma^\varepsilon \times Q \to \mathbb{N}$ (where $\mathbb{N}$ denotes the positive integers),
4. $q_0 \in Q$ is the initial state, and

5. $F \subseteq Q$ is the set of accept states.

Let $|A|_Q = |Q|$ be the number of states and $|A|_\delta = \sum_{q_1,q_2 \in Q, \alpha \in \Sigma^\varepsilon} \delta(q_1, \alpha, q_2)$ the number of transitions in A.

A *run* of an NFA on a given input string is a sequence of states, input symbols and transition labels from $\mathbb{N}$, traversed by the matcher while attempting to find a match. The formal definition of a run is given next. As a notational convenience let $\pi_\Sigma(w)$ (w is a string over Σ' where $\Sigma \subseteq \Sigma'$) represent the word $b_1 \dots b_n \in \Sigma^*$, where $b_i = a_i$ if $a_i \in \Sigma$, and $b_i = \varepsilon$ otherwise. Informally, $\pi_\Sigma(w)$ denotes the word obtained from w by removing all symbols not in Σ.

Definition 2.3 ([12])**.** For an NFA $A = (Q, \Sigma, q_0, \delta, F)$ and $w \in \Sigma^*$, a *run* of A on w is a string $r = s_0\alpha_1(j_1)s_1 \dots s_{n-1}\alpha_n(j_n)s_n$, where $s_0 = q_0$, $s_i \in Q$, $\alpha_i \in \Sigma^\varepsilon$ and $j_i \in \mathbb{N}$ such that $\delta(s_i, \alpha_{i+1}, s_{i+1}) \geq j_{i+1}$ for $0 \leq i < n$, and $\pi_\Sigma(r) = w$. A run is accepting if $s_n \in F$.

Figure 2.1 shows a state diagram of an NFA over the alphabet $\{\mathtt{a}\}$ that recognises the language $\{\varepsilon\} \cup \{\mathtt{a}^n \mid n \in \mathbb{N} \text{ and } n \text{ is divisble by 2 or 3}\}$.

Note that we define the transition function δ in such a way to allow for *parallel transitions*, that is multiple transitions between the same two states on the same transition symbol, or even multiple ε-transitions. Explicitly, $\delta(q_1, \alpha, q_2) = n$, where $n > 0$, implies there are n transitions from state q_1 to q_2 on input α (and no transitions if $n = 0$). We number these multiple transitions from 1 to n. When describing a path in an NFA we distinguish between parallel transitions using the notation $\alpha(i)$, where α is the input symbol of the corresponding transition and i denotes the ith transition on α between the two states under consideration. For example, $p \xrightarrow{\alpha(2)} q$ indicates the second transition on symbol α from state p to q. We will also use from time to time subscripts to number parallel transitions, for example, $p \xrightarrow{\varepsilon_i} q$ denotes the ith ε-transition from p to q.

We extended the definition of an NFA to allow for parallel transitions, since we are not only interested in the language accepted by an NFA, but also in the number of paths that can be used to accept a given word.

When drawing an NFA, parallel transitions will be indicated by either drawing the multiple transitions explicitly, or by adding multiple copies of a given label to a transition. Figure 2.2 illustrates these two methods.

Sometimes, we will work with automata for which parallel transitions are not possible. In these cases and all cases in which parallel transitions are either impossible or irrelevant, we will use the standard notation for transitions: $\delta : Q \times \Sigma^\varepsilon \to \mathcal{P}(Q)$.

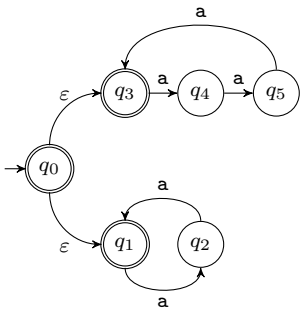

Figure 2.1: A state diagram representing an NFA that recognises the language over the alphabet {a} of strings with length a multiple of two or three.

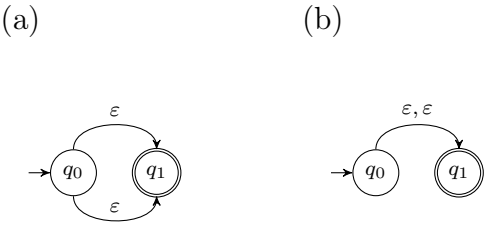

Figure 2.2: An illustration of the different ways of drawing parallel transitions, by either (a) drawing each transition explicitly, or (b) adding multiple labels on the same transition.

2.4 Regular Operations

If a language is recognised by an NFA, it is said to be a *regular language*. Various operations, known as *regular operations*, preserve regularity. Next we define three such operations.

Definition 2.4. Assume A and B are languages. Then union, concatenation and Kleene star are defined as follows.

1. Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
2. Concatenation: $A \cdot B = \{xy \mid x \in A \text{ and } y \in B\}$.
3. Kleene star: $A^* = \{\varepsilon\} \cup \{x_1 x_2 \dots x_k \mid k \geq 1 \text{ and each } x_i \in A\}$.

The basic idea behind the union operation is simply to combine all the strings in the operand languages into a new language. With concatenation a new language is formed by concatenating strings in the first language with strings in the second. Lastly, the Kleene star operation on A defines the language $(\cup_{i=1}^{\infty} A^i) \cup \{\varepsilon\}$, where A^i denotes A concatenated with itself i times.

2.5 Regular Expressions

The expressions built inductively from $\emptyset, \varepsilon, a \in \Sigma$ and (some of) the regular operations, are called *regular expressions*. The language defined by a regular expression E, denoted by $\mathcal{L}(E)$, is the (possibly empty) set of strings obtained by recursively defining $\mathcal{L}(F)$ for each subexpression F of E.

Definition 2.5. A regular expression R is either:

1. a, for some a in the alphabet Σ, where $\mathcal{L}(a) = \{a\}$,
2. ε (the empty string), where $\mathcal{L}(\varepsilon) = \{\varepsilon\}$,
3. $\emptyset$, where $\mathcal{L}(\emptyset) = \emptyset$,
4. $R = (R_1 \mid R_2)$, where R_1 and R_2 are regular expressions, where $\mathcal{L}(R) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,
5. $R = (R_1 \cdot R_2)$, where R_1 and R_2 are regular expressions, where $\mathcal{L}(R) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, or
6. $R = (R_1{}^*)$, where R_1 is a regular expression, where $\mathcal{L}(R) = \mathcal{L}(R_1)^*$.

When evaluating regular expressions (i.e., determining the language defined by regular expressions), * (Kleene star) takes precedence over $\cdot$ (Concatenation) and concatenation takes precedence over $\mid$ (Union). We can override the normal precedence of the regular operations by using parentheses. Regular expressions can further be compressed by writing E_1E_2 instead of $E_1 \cdot E_2$. We shall use the convention of using E_1^k to denote the expression E_1 concatenated with itself k times. Let $|E|$ be the number of subexpressions in regex E, or equivalently, the number of symbols in E, excluding parentheses. We number the subexpressions in E in a depth-first preorder, denoting by $E(i)$ subexpression i (e.g. for $E = ((\mathtt{a} \cdot \mathtt{b}) \mid \mathtt{b})^*$ we have $E(1) = E$, $E(3) = \mathtt{a} \cdot \mathtt{b}$ and $E(4) = \mathtt{a}$).

2.6 Regular Expression to NFA Construction

Regular languages are closed under regular operations, so by definition the language denoted by a regular expression must be a regular language. We stated in Section 2.4 that regular languages are those recognised by NFAs. Consequently, the language denoted by a regular expression can be recognised by some NFA. To obtain an NFA recognising the language denoted by a regular expression is fairly straightforward. Using Definitions 2.6 to 2.11 we can construct an NFA recognising the language denoted by a given regular expression. This can be achieved by starting at a base case (Definitions 2.6 to 2.8) and building up the rest of the expression by repeatedly applying Definitions 2.9 to 2.11 as needed.

Definition 2.6. (NFA recognising a single alphabet symbol a')
Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA where:

1. $Q = \{q_0, q_1\}$,
2. Σ is an alphabet containing a',
3. $\delta(q_0, a', q_1) = 1$ (and δ is 0 otherwise), and
4. $F = \{q_1\}$.

Definition 2.7. (NFA recognising ε)
Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA where:

1. $Q = \{q_0, q_1\}$,
2. $\delta(q_0, \varepsilon, q_1) = 1$ (and δ is 0 otherwise), and
3. $F = \{q_1\}$.

Definition 2.8. (NFA recognising $\emptyset$)
Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA where:

1. $Q = \{q_0\}$,
2. δ is always 0, and
3. $F = \emptyset$ (there are no accept states).

Definition 2.9. (Union)
Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be NFAs over the same alphabet Σ. We define $A = (Q, \Sigma, \delta, q_0, F)$ as:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$ (where we assume the sets in the union are disjoint and $q_0 \notin (Q_1 \cup Q_2)$),
2. if $q, q' \in Q$ and $a \in \Sigma^\varepsilon$, then

$$\delta(q, a, q') = \begin{cases} \delta_1(q, a, q') & \text{if } q, q' \in Q_1 \\ \delta_2(q, a, q') & \text{if } q, q' \in Q_2 \\ 1 & \text{if } q = q_0, a = \varepsilon \text{ and } q' = q_1 \\ 1 & \text{if } q = q_0, a = \varepsilon \text{ and } q' = q_2 \end{cases}$$

3. q_0 is the initial state, and
4. $F = F_1 \cup F_2$.

We have that $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.

Definition 2.10. (Concatenation)
Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be NFAs over the alphabet Σ. We define $A = (Q, \Sigma, \delta, q_0, F)$ as:

1. $Q = Q_1 \cup Q_2$ (where we assume the sets in the union are disjoint),
2. if $q, q' \in Q$ and $a \in \Sigma^\varepsilon$, then
$$\delta(q, a, q') = \begin{cases} \delta_1(q, a, q') & \text{if } q, q' \in Q_1 \\ \delta_2(q, a, q') & \text{if } q, q' \in Q_2 \\ 1 & \text{if } q \in F_1, a = \varepsilon \text{ and } q' = q_2 \end{cases}$$
3. $q_0 = q_1$, and
4. $F = F_2$.

We have that $\mathcal{L}(A) = \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$.

Definition 2.11. (Kleene star)
Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an NFA. We define $A = (Q, \Sigma, \delta, q_0, F)$ as:

1. $Q = \{q_0\} \cup Q_1$ (where we assume that $q_0 \notin Q_1$),
2. if $q, q' \in Q$ and $a \in \Sigma^\varepsilon$, then
$$\delta(q, a, q') = \begin{cases} \delta_1(q, a, q') & \text{if } q, q' \in Q_1 \\ 1 & \text{if } q \in F_1, a = \varepsilon \text{ and } q' = q_1 \\ 1 & \text{if } q = q_0, a = \varepsilon \text{ and } q' = q_1 \end{cases}$$
3. $F = \{q_0\} \cup F_1$.

We have that $\mathcal{L}(A) = \mathcal{L}(A_1)^*$.

2.7 Ambiguity of NFA

An NFA is said to be *ambiguous* if some string can be accepted by traversing two or more distinct paths of states and transitions. The *degree of ambiguity* of a string w (in an NFA M) is the number of accepting paths for w in M. Similarly, the degree of ambiguity of an NFA M is defined as the maximum number of ways in which any string in the language can be accepted. Let us use the NFA shown in Figure 2.1 to explain the concept of ambiguity. The strings described by $\mathtt{a}^n$ for which n is a multiple of either 2 or 3, but not both, are not ambiguous. When n is a multiple of 2, but not 3, the string can only be accepted by traversing the q_1, q_2 loop. Conversely, when n is a multiple of 3, but not 2, the string can only be accepted by traversing the q_3, q_4, q_5 loop. The strings $\mathtt{a}^m$ where m is a multiple of both 2 and 3 are indeed ambiguous. Since each of these strings can only be accepted by either the q_1, q_2 or q_3, q_4, q_5 loop, these strings have a degree of ambiguity of 2. Furthermore, since there are no other strings in $\mathcal{L}(M)$, the NFA M has a degree of ambiguity of 2.

2.7.1 Infinite Degree of Ambiguity

For some NFAs there is no upper bound on the number of ways in which all input strings can be accepted. In essence, for any string that can be accepted in m ways, there is another string that can be accepted in more than m ways. In these cases we say that these NFAs have *infinite degree of ambiguity (IDA)* [13].

For NFAs with IDA, we are exclusively interested in those NFAs for which the degree of ambiguity grows in relation to the length of the input string. For some NFAs, a single input string (of finite length) can be accepted in an infinite number of ways. This usually occurs for NFAs with loops consisting only of ε-transitions (called ε-loops). We disregard these NFAs when discussing degree of ambiguity and therefore, when referring to the degree of ambiguity of an NFA, it can be assumed the NFA does not have ε-loops.

To illustrate the growth of the degree of ambiguity in relation to the length of the input string, suppose M is an NFA with IDA. If the degree of ambiguity of M has an order of growth $O(n)$, with n being the length of the input string, we say M has IDA of degree 1. Intuitively, IDA of degree (at least) 1 occurs if M has two states p_0 and q_0, each with a self-loop on the same input string w, as well as the property that q_0 can be reached from p_0 while reading w [14]. Figure 2.3 shows an abstract NFA with IDA of degree at least 1. The dashed transitions in the abstract NFA represent substructures that recognise $\mathcal{L}(E)$, where E is the expression on the label of the dashed transition. For the sake of simplicity, we assume that the expressions on the labels of the dashed transitions, and the substructures recognising them, are unambiguous. To see why this abstract NFA has IDA of degree at least 1, suppose the expressions $E_{0,0}$, $E_{0,1}$ and $E_{0,2}$ recognise at least one common input string w, that is $w \in \mathcal{L}(E_{0,0}) \cap \mathcal{L}(E_{0,1}) \cap \mathcal{L}(E_{0,2})$. The input string ww has degree of ambiguity 2, since it can be recognised via the paths $p_0 \xrightarrow{w} p_0 \xrightarrow{w} q_0$ or $p_0 \xrightarrow{w} q_0 \xrightarrow{w} q_0$. Similarly, the input string www has a degree of ambiguity of 3, since the accepting paths are $p_0 \xrightarrow{w} p_0 \xrightarrow{w} p_0 \xrightarrow{w} q_0$, $p_0 \xrightarrow{w} p_0 \xrightarrow{w} q_0 \xrightarrow{w} q_0$ and $p_0 \xrightarrow{w} q_0 \xrightarrow{w} q_0 \xrightarrow{w} q_0$. In general for w^n, there is a linear number of ways to divide the n substrings w between the two loops in the abstract NFA.

If the degree of ambiguity of M has an order of growth $O(n^2)$, we say M has IDA of degree 2. Figure 2.4 shows an abstract NFA with IDA of degree at least 2. Suppose $w_0 \in \mathcal{L}(E_{0,0}) \cap \mathcal{L}(E_{0,1}) \cap \mathcal{L}(E_{0,2})$, $w_1 \in \mathcal{L}(E_{1,0}) \cap \mathcal{L}(E_{1,1}) \cap \mathcal{L}(E_{1,2})$ and $v_0 \in \mathcal{L}(F_0)$. In this case, for the input string $w_0^n v_0 w_1^n$, there is a quadratic number of ways, in n, to divide the w_0 and w_1 substrings between the four loops in the abstract NFA. Specifically, for each of the n ways of dividing the w_0 substrings between the p_0 and q_0 loops, there are n ways of dividing the w_1 substrings between the p_1 and q_1 loops. Therefore, the abstract NFA has IDA of degree of at least 2.

This concept can be generalised to an NFA with IDA of degree n, as shown in Figure 2.5.

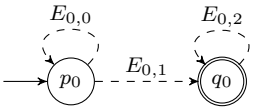

Figure 2.3: An abstract NFA with IDA of degree at least 1.

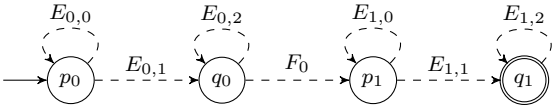

Figure 2.4: An abstract NFA with IDA of degree at least 2.

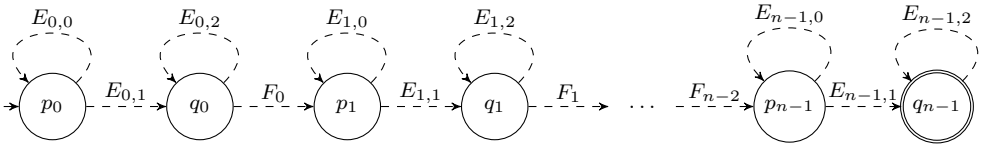

Figure 2.5: An abstract NFA with IDA of degree at least n.

2.7.2 Exponential Degree of Ambiguity

In Section 2.7.1 we have seen that it is possible to make the degree of IDA arbitrarily large by adding more of the structures with states p_i and q_i, having the self-loops on $E_{i,0}$ and $E_{i,2}$ respectively and joined with the transition through $E_{i,1}$. If there existed an NFA with an infinite number of these structures, there could be no polynomial with degree high enough to describe the rate of growth of the degree of ambiguity in the length of the input string. We can simulate such an NFA by adding a loop, so that at least one pair of p_i and q_i states can be traversed an arbitrary number of times — this is illustrated in Figure 2.6. In cases such as these the rate of growth of the degree of ambiguity is *exponential* in the length of the input string, i.e., the NFA has an *exponential degree of ambiguity (EDA)* [13].

Intuitively, exponential degree of ambiguity occurs when there is some state in the NFA such that this state can loop back to itself in more than one way, reading the same input string in each case. This can be illustrated with an abstract NFA such as the one shown in Figure 2.7. Suppose there exists some input string $w \in \mathcal{L}(E_{0,0}) \cap \mathcal{L}(E_{0,1})$. We refer to the self-loop for the expression $E_{0,0}$ as the top loop and for the expression $E_{0,1}$ as the bottom loop. For an input string such as w^n, every w substring can be matched by either the top loop or the bottom loop. Therefore, every w added to the input string will cause the number of ways to divide the w substrings between the top– and

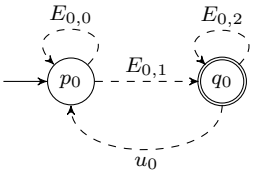

Figure 2.6: An abstract NFA with EDA.

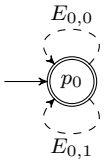

Figure 2.7: A more intuitive illustration of an abstract NFA with EDA.

bottom loop to double. In other words, the degree of ambiguity will double for every w appended to the input string. Consequently, the degree of ambiguity is exponential in the length of the input string and the NFA is exponentially ambiguous.

2.7.3 Formal Definition of Degree of Ambiguity

Next we formally define infinite and exponential degree of ambiguity.

Definition 2.12. The degree of ambiguity for $w \in \Sigma^*$, with respect to the NFA A (without ε-loops), denoted by $d_A(w)$, is the number of accepting runs on w in A. The degree of ambiguity of A is the maximum degree of ambiguity over all $w \in \Sigma^*$, which might be infinite, in which case we say A has *infinite degree of ambiguity* (IDA). When A has IDA, we consider the rate at which the maximum number of accepting runs grows in proportion to the length of the input strings. This rate might be exponential, described by saying A has *exponential degree of ambiguity* (EDA), or polynomial, described as A being *polynomially ambiguous*.

2.8 Conclusion

In this chapter we defined relevant concepts in formal language theory — we defined regular languages and their relation to NFAs. We explained what it means for an NFA to have either finite or infinite ambiguity. Using these concepts we will present a decision procedure for when regexes are vulnerable.

Chapter 3

Regex Fundamentals

3.1 Introduction

Even though regexes originate from a well-defined mathematical model, as described in Chapter 2, in practice regexes have been extended to increase their pattern matching capability further. In this chapter we investigate features and syntax found in practical regex matching engines. Specifically, we focus on the regex engine found in Java.

3.2 Regexes in Practice

As discussed in Chapter 2, the theoretical study of regular expressions forms part of the field known as formal language theory. A formal language comprises a set of strings or words over symbols known as an alphabet. Dependent on the class of formal languages under consideration, a specific formalism is used to specify the strings in a given formal language of interest.

Similar to theoretical regular expressions, regexes in practice typically operate over an alphabet consisting of all symbols in some character set, for example the US-ASCII character set, and a specific regex describes the language of words matched by the regular expression under consideration. We say a regex *matches* an input string if it is contained within the language of words described by the regex under consideration. Note that in some cases a regex is said to match an input string if there is a substring of the input string that is matched by the regular expression — this is referred to as *submatching*. In other cases a regex only matches an input string if the entire input string is matched by the regular expression, referred to as *exact matching*. We focus mainly on exact matching, but it is possible to convert from one to the other, as shown in Section 3.6.

On a high level, regexes are sequences of symbols used to describe patterns. These patterns allow for a more general search through text than simply searching for specific words verbatim. As a simple example, the regex `colou?r` can

be used to search for occurrences of the words `color` and `colour` in a piece of text, when the reader is unsure whether the author used American or British spelling.

3.3 Regular Expression Extensions

As regular expressions became more popular as a pattern matching tool, extensions were added to regular expression matching engines to increase the class of possible patterns that can be specified, and also the ease with which these patterns can be specified. Regular expressions extended with these features are typically referred to as *regexes* or *extended regexes*. We only use the term *regular expression* when we are specifically referring to the mathematical model defined in Section 2.5.

Regexes developed into multiple different standards, called *flavours* of regexes. We focus specifically on the flavour used in the Java programming language, implemented in the `java.util.regex` package.

Two of the most common extensions built into regexes are the *question mark operator* and the *plus operator*. The expression E? may be considered as an abbreviation for $(E \mid \varepsilon)$, whereas E^+ denotes EE^*. For example, the regex `Mon(day)?` can be used to match either `Monday` or `Mon` if, for example, a search is required for both the word *Monday* and its abbreviation in a piece of text. The plus operator can for example be used to search for positive, integral powers of ten in a list of numbers with the regex `10`$^+$, as it will match `10`, `100`, `1000` and so on.

Another extension is the *interval quantifier*, which ensures that a regex matches with the operand subexpression any number of times within a specified range. Reusing the previous example of searching for positive, integral powers of ten in a list of numbers, the interval quantifier makes it possible to place bounds on the powers of ten to search for, for example, `10{`$2, 4$`}` will match the strings `100`, `1000` and `10000`.

All of the above mentioned extensions can be represented in equivalent ways with regular expressions. These particular extensions merely provide notational convenience. Especially in the case of the interval quantifier, explicitly specifying the different options represented might not be practical when the interval is large. Indeed, rewriting a regex as a regular expression can increase the size exponentially (in terms of the number of symbols used to represent the regex and regular expression). Consider the regex `a{10}`, which is represented with 5 symbols. The equivalent regular expressions is `aaaaaaaaaa`, which requires 10 symbols to represent. In general, a regular expression equivalent to E`{`n`}` will be exponentially longer.

Not all regexes can be converted to equivalent regular expressions. One extension, called a *backreference*, instructs a matcher to match the same substring as was matched by a previous subexpression. This cannot be achieved by

a regular expression, since it allows the specification of nonregular languages. To illustrate this, consider the regex (a*)b\1, which matches any number of a's, followed by a b, and again followed by the same number of a's as before. The language matched by this regex is not regular, since an NFA for this language will be required to store the exact number of a's that was matched before the b, which is not possible.

3.4 Regex Basics

In this section we look at how regexes are constructed and used in practice. Most characters in a regex instruct the matcher to match the specific specified character in a given input string. Specifically, the regex abc instructs the matcher to match the symbol a followed by b, followed by c. Note that this regex consists of the subexpressions a, b and c, each indicating a match with the corresponding symbol. By writing these regexes consecutively, we indicate that in the input string these symbols should also appear in succession. This corresponds to concatenation as defined in Chapter 2. Regex concatenation is obviously not limited to single symbol subexpressions. That is, for the regexes E_1 and E_2, their concatenation E_1E_2 matches any concatenation of strings matched by E_1 and E_2.

Some characters, called *metacharacters*, have a different meaning than simply indicating a match with the corresponding symbol. The metacharacters make it possible to describe general patterns either by performing operations on subexpressions, or by performing grouping to create such subexpressions. Some of the most frequently used metacharacters are broadly described in the next few subsections. For an in-depth discussion consult [15].

3.4.1 Alternation

Alternation in regexes gives the matcher a choice between matching either with the first or with the second operand subexpression. For example, $(E_1 \mid E_2)$ will match an input string with E_1 or E_2. In the case of certain regex matchers, such as the one used in Java, the matcher first attempts to match an input string with E_1 and only attempts matching with E_2 if a match with E_1 is impossible. In essence, using our previous example of the regex $(E_1 \mid E_2)$, matching with the subexpression E_1 is assigned a higher priority than with E_2.

3.4.2 Quantifiers

Quantifiers in regexes allow for a subexpression to be used a varying number of times while matching. The most common quantifier is the *zero-or-more* quantifier denoted by a star. This is referred to as the Kleene star operator (or

Kleene star quantifier) and performs a similar role as the Kleene star operator found in regular expressions (as discussed in Chapter 2). The Kleene star operator allows the matcher to use the operand subexpression zero or more times in succession while matching. For example, for a regex E, the regex E^* matches ε (the empty string), all strings matched by E, by E^2 (that is E concatenated with itself), by E^3, etc.

Other quantifiers include the *zero-or-once* quantifier, denoted with a question mark as in $E?$ (which matches either ε or uses E for matching), and the *once-or-more* quantifier, denoted by a plus sign, as in E^+ (matching with E, or E^2, or E^3, etc.). We refer to the zero-or-once and the once-or-more quantifier as the *question mark* and *plus quantifier* respectively.

The interval quantifier forces the operand subexpression to be used repeatedly, a number of times in a specified range, in succession. For example, $E\{l, h\}$ matches with E^h, or E^{h-1}, ..., or E^l, with $l \leq h$.

Any quantifier with finite bounds can be rewritten using only alternation. For example, $E_1?$ is equivalent to $(E_1 \mid \varepsilon)$. Similarly, the regex $E_1\{l, h\}$ is equivalent to $(E_1^h \mid E_1^{h-1} \mid \ldots \mid E_1^l)$.

As with alternation, certain matches are prioritised over others. In the default case with $E\{l, h\}$ for example, matching with E^i is assigned a higher priority than with E^{i-1}, assuming $l < i \leq h$. This holds true for the implicit bounds of the Kleene star quantifier, question mark quantifier and the plus quantifier as well. This prioritisation can also be reversed, which is discussed in greater detail in Section 3.4.2.1.

3.4.2.1 Greedy and Lazy Quantifiers

Quantifiers can be either *greedy* or *lazy*. For a greedy quantifier, after matching a nonempty substring using the operand of the quantifier, the matcher will rather match with the operand again, than with subexpressions successive to the quantifier. Note that this is done while remaining in the bounds of the quantifier. For the greedy question mark quantifier $E?$, the matcher will rather match with E, than with the implicit ε. Similarly, for the greedy interval quantifier $E\{l, h\}$, the matcher will be forced to use E for matching at least l times, after which it will rather match with E again, rather than using subsequent subexpressions (assuming E has been used fewer than h times).

For a lazy quantifier, the matcher does the opposite, i.e., the matcher rather matches with successive subexpressions instead of using the operand of the lazy quantifier again. In Java, quantifiers are altered to be lazy by appending a question mark symbol after the quantifier. That is $E^{*?}$, $E??$, $E^{+?}$ and $E\{m, n\}?$ denote the lazy Kleene star, question mark, plus and interval quantifiers respectively.

Greedy and lazy quantifiers can be explained using the concept of priorities, which was mentioned in Section 3.4.1. Similarly to alternation in which the left side operand subexpression is matched with a higher priority than the right

side operand, greedy quantifiers prioritise matching again with the operand subexpression higher than matching with successive subexpressions, while lazy quantifiers do the opposite.

It may be tempting to think that if a greedy quantifier is applied to a subexpression, a new subexpression is obtained which matches longest possible substrings (when the starting point of the submatch is specified), but this is not always true. To see this, consider the regex `(a|ab)*b?`. For the input string `ab`, the matcher will elect to match `a` with the left operand of the first alternation. After the failure to match `b` with the left operand of the `(a|ab)` subexpression, the matcher continues and matches `b` with the `b?`. This is done in preference to releasing the matched symbol `a` and rematching the entire input string with the right side of the first alternation, i.e., `ab`. Consequently, the subexpression with the greedy Kleene star quantifier matches the shorter substring `a` with a higher priority than longer substring `ab`.

Note that changing a greedy quantifier to a lazy quantifier (or conversely), does not modify the set of strings a given regex can match, but instead only changes what substring is matched by which subexpression.

3.4.3 Character Classes

Character classes provide a compact way to express a set of single symbols that can all be matched. For example, the character class `[abc]` can match either of the symbols `a`, `b` or `c`. Note, since character classes only ever match a single symbol, symbols written next to each other in a character class denotes their union, rather than concatenation. Character classes also allow for the expression of a range of characters in a concise way. For example, the character classes `[a-c]` and `[abc]` are equivalent. A range of characters can be specified between any two characters, as long as the byte representation of the first character is smaller or equal to that of the second. For example, the character range `[`x`-`y`]` specifies all characters with a byte representation between that of x and y (inclusively).

Java contains multiple predefined character classes — some of these are listed in Table 3.1. Note, we only give a few examples of the predefined character classes for specifying character properties (denoted by `\p{...}`). In total there are more than 600 ways to specify character properties and many can be found in [15]. Java also defines a *wildcard* symbol which acts as a character class matching any single symbol. We use the symbol `.` to denote the wildcard symbol.

3.4.3.1 Intersection of Character Classes

Intersection can be performed in character classes with the `&&` operator. For example, `[abc&&cde]` matches only `c`, as it is the only symbol in both `[abc]` and

Table 3.1: Some of the predefined character classes found in Java.

Sequence	Description	Negated
`\d`	Digits	`\D`
`\w`	Word characters	`\W`
`\s`	White space characters	`\S`
`\p{Lower}`	Any lowercase Latin Alphabet character	`\P{Lower}`
`\p{Digit}`	Any Arabic digit character	`\P{Digit}`
`\p{Punct}`	Any punctuation character	`\P{Punct}`

`[cde]`. Note that the union in character classes, mentioned in Section 3.4.3, takes precedence over intersection.

3.4.3.2 Negation of Character Classes

A character class can be negated so that it matches any single symbol not in the original character class. This is done by adding a caret symbol `^` at the start of the character class. For example, `[^abc]` will match any single symbol, except for `a`, `b` or `c`.

3.4.3.3 Nested Character Classes

Character classes may contain other character classes (referred to as *nested* character classes). This becomes especially useful when used in combination with intersection and negation. For example, `[[a-z]&&[^aeiou]]` matches any single symbol corresponding to a lowercase Latin alphabet letter, except for vowels.

3.4.4 Grouping

It is possible to group subexpressions by placing parentheses around the appropriate subexpressions, and then applying for example alternation and quantifiers to these grouped subexpressions. In some cases grouping has the side effect of indicating to the regex engine to store the substring of the input string that was matched by the subexpression contained within the group. Groups that store the matched substring are referred to as *capturing groups*, and even though it is possible to specify noncapturing groups, we will assume all groups are capturing.

Capturing groups make it possible to perform certain shallow parsing tasks. For example, `(1[012]|[1-9]):([0-5][0-9])(am|pm)` can be used to parse input strings in 12-hour time format. The hour is stored by the first capturing group `(1[012]|[1-9])`, the minutes by the second group `([0-5][0-9])`, and time period by the third `(am|pm)`. The contents of a capturing group can be accessed after the match, or during the match as in the case of backreferences (see

Section 3.5). In the Java programming language, after a match, the contents of a specific capturing group is accessed using the method `matcher.group(`i`)`, with `matcher` an instance of the `Matcher` class used to perform the match and i a positive integer corresponding to the ith capturing group. Note that capturing groups are ordered by the position of their left (or opening) parentheses. The capturing group with the leftmost opening parenthesis is capturing group number 1.

Storing substrings matched by subgroups requires that the regex matcher keeps track of what was matched by every capturing group of the regex, which is straightforward for a backtracking matcher, but is nontrivial to implement in other matchers. Thus capturing groups are typically only implemented in backtracking matchers, with RE2 being the exception [9].

3.4.5 Atomic Groups and Possessive Quantifiers

Atomic groups indicate to the matcher that any substring matched with the subexpression contained in the atomic group, should be regarded as a unit. This means that as soon as the atomic group has matched a substring, any stored information regarding other ways the atomic group subexpression could have matched any prefix of the string remaining to be matched, is disregarded. Also, possessive quantifiers are a notational convenience for quantifiers in atomic groups. Marking a quantifier as possessive indicates that the quantifier and its operand subexpression are grouped in an atomic subgroup. In other words, the regex (E^{*+}), where E^{*+} denotes the possessive Kleene star quantifier applied to the regex E, can be expressed with an atomic group as $(?>E^{*})$, where $(?>\ldots)$ denote an atomic group. It follows similarly for the other possessive quantifiers.

In [15], a motivating example is provided for the use of atomic groups. The regex in this example is required to truncate decimal numbers at the second digit after the decimal point (i.e., the second digit should be kept), except when the third digit is nonzero, in which case the truncated number should include three digits after the decimal point. A regex serving this purpose would need to match decimal numbers with three or more digits to the right of the decimal point. Using capturing groups (see Section 3.4.4), one could replace the matched decimal number with another decimal number that has only two or three digits to the right of the decimal point. This replacement is done in a way depending on if the third digit after the decimal point in the matched number is zero or not. A regex similar to $(\backslash d^{+}\backslash .\backslash d\backslash d[1\text{-}9]?)\backslash d^{*}$ is suggested for this purpose. The capturing group $(\backslash d^{+}\backslash .\backslash d\backslash d[1\text{-}9]?)$ matches any decimal number with two digits after the decimal point, as well as those with a third nonzero digit. Digits that are three (if the third is zero, or four if the third is nonzero) or more positions after the decimal point is matched by $\backslash d^{*}$, which is not captured, and can thus be removed. Even though this is sufficient for our purpose, we consider next what happens when the regex matches

0.125. The capturing group matches the entire decimal number, so effectively 0.125 will be replaced with itself. One might be tempted to improve the efficiency of the incorporating find-and-replace code by modifying the regex to be `(\d+\.\d\d[1-9]?)\d+`. This modification restricts matches to decimal numbers with digits that will be removed when substituting the original number with the contents of the capturing group. But with input 0.125, the subexpression `[1-9]?` is optional and the subexpression `\d+` not, and thus the last digit 5 of 0.125 will be matched by `\d+`. Consequently, it will not be included in the capturing group and so it will be removed in the substitution, rounding 0.125 to 0.12. This can be fixed by including the `[1-9]?` subexpression in an atomic group, i.e., using `(\d+\.\d\d(?>[1-9]?))\d+`. In doing so, if `[1-9]?` is capable of matching part of the decimal number, it can never be released to match the final `\d+` instead.

Further motivation for atomic groups and possessive quantifiers is provided by the fact that it may allow a match to fail faster. Suppose for example the regex `[0-9]+\.[0-9]+` is used to search for numbers with at least some digits after the decimal point, from a given list of numbers. Since the first subexpression `[0-9]+` can match any number of digits (as long as it matches at least one), for every nonnegative integer encountered in the list of numbers, say `1000`, it will first match all digits in `1000` (as the plus quantifier matches greedily). Since `1000` is not followed by a decimal point, the first subexpression `[0-9]+` will release the last `0` to match `100`, but again it will fail due to the missing decimal point. This process will be repeated until the first subexpression `[0-9]+` matches only one digit and the matcher finally decides that the input can not be matched by the regex (this matching process is explained in more detail in Section 4.4). This takes unnecessarily many steps, as the match will obviously fail due to the missing decimal point. If the regex is converted to `[0-9]++\.[0-9]++` , to make the plus quantifiers possessive, the match will fail faster. After the nonnegative integer `1000` is matched with the first `[0-9]++`, the matcher disregards the information that fewer digits could also have been matched. Consequently, when it does not find the decimal point, the entire match fails immediately. Note that the regex `[0-9]++\.[0-9]++` can be converted into an equivalent regex using atomic groups: `(?>[0-9]+)\.(?>[0-9]+)`. In Chapter 11 atomic groups and possessive quantifiers are investigated more thoroughly as a means to prevent excessive matching time vulnerabilities in regexes.

3.4.6 Lookaround Assertions

Lookaround assertions allow for the placement of restrictions on prefixes (or suffixes) starting (or ending) at specific positions in the input string. There are four main types of lookaround assertions, namely positive or negative lookahead or lookbehind assertions.

Positive lookahead assertions assert that at some position in the input string, an operand subexpression should match a prefix of the suffix starting at that point. This is usually achieved by matching a prefix of the suffix of the input string with the lookaround assertion, but if this match is successful, the matching position in the input string is reset to the position before matching with the lookahead assertion started. For example the regex $(\texttt{?=}E_1)E_2$, where $(\texttt{?=}E_1)$ denotes a positive lookahead assertion using E_1, describes the language $\mathcal{L}(E_1\Sigma^*) \cap \mathcal{L}(E_2)$.

Similarly, positive lookbehind assertions assert that at some position in the input string, a suffix of the currently matched substring, should be matched by a given regex. For example the regex $(E_1(\texttt{?<=}E_2))$, where $(\texttt{?<=}E_2)$ denotes a positive lookbehind assertion on E_2, matches an input string precisely when some suffix of a string matched by E_1, can also be matched by E_2.

The negative lookahead (denoted by $(\texttt{?!}E_1)$) and negative lookbehind (denoted by $(\texttt{?<!}E_2)$) assertions function similarly as their positive counterparts, but instead assert that the operand subexpression *should not* match a prefix or suffix at some position.

Java contains some predefined lookaround assertions, such as the word-boundary assertion `\b`. The word-boundary assertion specifies that the matcher should be between a word character and nonword character in the input string. Other predefined lookaround assertions include the string anchor assertions. The caret symbol (^) requires that the matcher should be at the beginning, whereas the dollar symbol ($) specifies that the matcher should be at the end of the input string.

3.5 Backreferences

Placing a backreference in a regex indicates that the substring matched by some specified capturing group, is required to be matched again at the position in which the backreference is placed. The capturing group to be used for the backreference is denoted by using the syntax $\backslash i$ in the regex, where i is the ith capturing group.

For example, $\texttt{[0-9]}^+\texttt{.\d}^*\texttt{(\d}^+\texttt{)\1}^+$ can be used to match recurring decimal numbers, such as $0.\overline{3}$, $0.8\overline{1}$ and $0.0\overline{45}$. The subexpression $\texttt{\d}^+$ captures some sequence of numbers matched in the input string and the backreference $\texttt{\1}^+$ instructs the matcher to match this sequence again, one or more times.

Another motivating example for backreferences is found in [15], where a regex similar to $\texttt{([a-zA-Z]}^+\texttt{)\1}$ is used to find occurrences where a word is repeated immediately.

3.6 Exact Matching and Submatching

In some cases an input string is said to be matched by a regex if the entire input string is matched — this is referred to as *exact matching*. In other cases, the input string is matched if some substring is matched, referred to as *submatching*. It is possible to do exact matching by using submatching and vice versa. Suppose that E is used for submatching and we want to replicate submatching behaviour by using exact matching. We achieve this by using $\texttt{.}^{*?}E\texttt{.}^{*}$ in exact matching. Conversely, if we want to use E for exact matching while assuming we have an algorithm or implementation for submatching, then we simply add the string anchors ^and $ (see Section 3.4.6) to E and use $\texttt{\^{}}E\texttt{\$}$.

3.7 Conclusion

In this chapter we discussed some of the extensions added to regular expressions, and commonly supported in regex matching engines. We have shown some extensions that provide only a notational convenience for regexes and that can be expressed in regular expressions, but other extensions allow regexes to match nonregular languages.

Chapter 4

Regex Matching

4.1 Introduction

In this chapter we explore the two main algorithms used by regex matchers, namely *subset construction matching* and *backtracking matching*. For backtracking matching we present a string-to-tree transducer model that allows us to abstract the matching time behaviour of an implemented regex matcher.

4.2 Subset Versus Backtracking Matching

To explain the difference between subset construction matching and backtracking matching, consider a regex $(\mathtt{w}E \mid \mathtt{w}F)$, where $\mathcal{L}(E) \cap \mathcal{L}(F) = \emptyset$. While attempting to match an input string, the matcher consumes the input string one symbol at a time and consequently does not know beforehand whether the match will ultimately be made with $\mathtt{w}E$, $\mathtt{w}F$, or not at all.

Subset construction matching traverses the different options in the regex concurrently. Suppose the input string $\mathtt{w}v$ is used where $v \in \mathcal{L}(E)$, such that v does not share a common nonempty prefix with any string in $\mathcal{L}(F)$. Therefore, the match will ultimately be made with the $\mathtt{w}E$ option in the alternation. While matching the prefix $\mathtt{w}$ of the input string the matcher will note that both sides of the alternation can still result in a possible match. Only when the matcher reaches the first symbol of v, will it decide that F is no longer a valid option (due to v not sharing any common nonempty prefix with a string in $\mathcal{L}(F)$) and only continue to match with E. Subset construction matching is explained in more detail in Section 4.3.

Even though subset construction matching is relatively fast in the worst-case, many regex engines opt to use backtracking matching, since backtracking matching allows for a much more intuitive implementation of some regex extensions (that might be impossible to implement with the subset construction), such as capturing groups and backreferences.

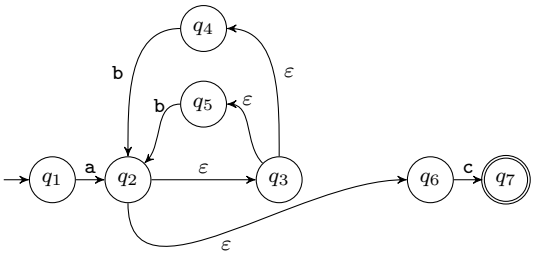

Figure 4.1: An NFA recognising the language $\mathcal{L}(E)$ where $E = \mathtt{a}(\mathtt{b} \mid \mathtt{b})^*\mathtt{c}$.

In the case of backtracking matching, for the same regex $(\mathtt{w}E \mid \mathtt{w}F)$ and input string $\mathtt{w}v$ used earlier, rather than matching with both sides of the alternation concurrently, a backtracking matcher will first attempt to match with $\mathtt{w}E$, and only if this is not possible, try to match with $\mathtt{w}F$. Note that a backtracking matcher prioritises matches made with certain subexpressions over others. In the above example, the matcher prioritises a match with E over a match with F. This method of matching is explained in more detail in Section 4.4.

This example generalises to any type of choice in a regex. For example, with a subexpression E^*, subset construction matching will traverse the input string as if it matched E zero or more times, while backtracking matching will match with E as many times as possible, noting each time that it could have continued without matching another E.

4.3 Subset Construction Matching

The main idea of subset construction matching is to keep a set of possible states in which the matcher currently could be, while reading the input string. If one of these possible states is an accept state when the entire input string has been read, the matcher accepts, otherwise it rejects. In doing this, the matching process is determinised.

Consider the NFA depicted in Figure 4.1. The matcher will start in the set of states containing only the initial state, namely $\{q_1\}$. Upon reading an `a` from the input string, the matcher will proceed to the set of states $\{q_2, q_3, q_4, q_5, q_6\}$. The matcher will remain in this set of states for every further `b` read from the input string, until a `c` is read, which will take the matcher to $\{q_7\}$. If the end of input is reached when transitioning to $\{q_7\}$, the matcher will accept.

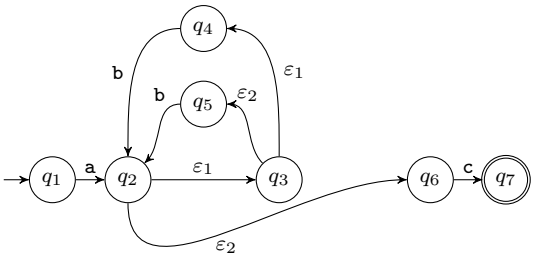

Figure 4.2: A pNFA recognising the language $\mathcal{L}(E)$ where $E = \mathtt{a}(\mathtt{b} \mid \mathtt{b})^*\mathtt{c}$.

4.4 Backtracking Matching

Backtracking matching, in essence, is a depth-first search on the NFA corresponding to the regex. The input string determines which symbol transitions are traversed. The order in which the nondeterministic transitions are traversed, is decided by the priority the matcher places on matches made with certain subexpressions over others. To simulate backtracking matching we need to ensure that for every input string and regex combination given to the matcher, the underlying NFA follows exactly one path through the transitions and states (including those backtracked from) to determine whether the input string is accepted or not.

Therefore, for backtracking matching we need to ensure that there is at most one accepting run (see Definition 2.3) in an NFA for a given input string, in contrast to traditional NFAs. We can achieve the desired property by using what is known as a *prioritised-NFA* (pNFA). When attempting to match an input string, a pNFA adds priorities on ε-transitions from a specific state, to indicate the order in which the backtracking matcher will traverse them. Furthermore, the matcher will not attempt a lower priority path if a match was found on a higher priority path. Figure 4.2 depicts a pNFA matching the same language as the NFA depicted in Figure 4.1. We formally define the concept of a pNFA next.

Definition 4.1 ([12])**.** A *prioritised nondeterministic finite-state automaton (pNFA)* is a 7-tuple $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, where if $Q := Q_1 \cup Q_2$, we have:

1. Q_1 and Q_2 are disjoint finite sets of states such that $Q \neq \emptyset$,
2. Σ is the finite input alphabet,
3. $\delta_1 : Q_1 \times \Sigma \to Q$ is the *deterministic* partial transition function,
4. $\delta_2 : Q_2 \to Q^*$ (where Q^* is the set of strings over Q) is the *nondeterministic prioritised* transition function,

5. $q_0 \in Q$ is the initial state, and
6. $F \subseteq Q_1$ is the set of accept states.

We use the notation $|\delta_2(p)|_q$ to denote the number of occurrences of q in $\delta_2(p)$.

Definition 4.2. Let nfa(A) for a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be the NFA $A' = (Q_1 \cup Q_2, \Sigma, \delta', q_0, F)$, where δ' is defined as follows. If $p, q \in (Q_1 \cup Q_2)$ and $\alpha \in \Sigma^\varepsilon$ (recall $\Sigma^\varepsilon = \Sigma \cup \{\varepsilon\}$), then

$$\delta'(p, \alpha, q) = \begin{cases} 1 & \text{if } p \in Q_1 \text{ and } \delta_1(p, \alpha) = q \\ n & \text{if } \alpha = \varepsilon \text{ and } p \in Q_2 \text{ and } n > 0 \text{ where } n = |\delta_2(p)|_q \end{cases}$$

That is, nfa(A) is the NFA obtained by removing the priorities on the ε-transitions of the pNFA A.

Informally, for a pNFA, the sets Q_1 and Q_2 contain the states to be used with the deterministic partial transition function δ_1 and the nondeterministic prioritised transition function δ_2, respectively. The function δ_1 allows the automaton to transition deterministically to the next state depending on the next input symbol and models the behaviour of a backtracking matcher when it consumes input symbols. The function δ_2 allows the automaton to select the next state nondeterministically from a sequence. The order in which a backtracking matcher will visit the subexpressions is modeled with the priorities placed on the transitions in δ_2.

4.4.1 Handling ε-transitions

When backtracking matching is used instead of subset construction matching, special care needs to be taken when handling ε-transitions, especially when encountering ε-loops.

In the model we use, no ε-transition is allowed to be used more than once without consuming an input symbol since its previous use. In practice, some matchers prevent infinite loops on ε-loops by restricting the reuse of only certain ε-transitions (See Appendix A). However, to simulate this accurately will result in a more complicated model.

Next we define *(accepting) paths* in pNFA.

Definition 4.3 ([12])**.** For a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, a *path* for $w \in \Sigma^*$ in A, is a run $s_0\alpha_1(j_1)s_1 \dots s_{n-1}\alpha_n(j_n)s_n$ of w in nfa(A), such that if $\alpha_i = \alpha_{i+1} = \dots = \alpha_{m-1} = \alpha_m = \varepsilon$, where $i \leq m$, then $(s_{k-1}, j_k, s_k) = (s_{l-1}, j_l, s_l)$, where $i \leq k, l \leq m$, implies $k = l$. In other words, a path is not allowed to repeat the same transition in a sequence of ε-transitions. For two paths $p = s_0\alpha_1(j_1)s_1 \dots s_{n-1}\alpha_n(j_n)s_n$ and $p' = s'_0\alpha'_1(j'_1)s'_1 \dots s'_{m-1}\alpha'_m(j'_m)s'_m$ we say that p is of higher priority than p', $p > p'$, if $p \neq p'$, $\pi_\Sigma(p) = \pi_\Sigma(p')$ and

Table 4.1: The corresponding ε-transition in flat(A) for every δ_2-path in A, where A is the pNFA in Figure 4.3(a).

δ_2-path in A	ε-transition in flat(A)
$q_0 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_1} q_2$	$q_0 \xrightarrow{\varepsilon_1} q_2$
$q_0 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_2} q_0 \xrightarrow{\varepsilon_2} q_3$	$q_0 \xrightarrow{\varepsilon_2} q_3$
$q_0 \xrightarrow{\varepsilon_2} q_3$	$q_0 \xrightarrow{\varepsilon_3} q_3$
$q_1 \xrightarrow{\varepsilon_1} q_2$	$q_1 \xrightarrow{\varepsilon_1} q_2$
$q_1 \xrightarrow{\varepsilon_2} q_0 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_1} q_2$	$q_1 \xrightarrow{\varepsilon_2} q_2$
$q_1 \xrightarrow{\varepsilon_2} q_0 \xrightarrow{\varepsilon_2} q_3$	$q_1 \xrightarrow{\varepsilon_3} q_3$

either p' is a proper prefix of p, or if k is the first index such that $(j_k)s_k \neq (j'_k)s'_k$, then $\delta_2(s_{k-1}) = \ldots s_k \ldots s'_k \ldots$ if $s_k \neq s'_k$, or $s_k = s'_k$ and $j_k < j'_k$. An accepting run for a pNFA A on w is the highest-priority path $p = s_0\alpha_1(j_1)s_1 \ldots \alpha_n(j_n)s_n$ such that $\pi_\Sigma(p) = w$ and $s_n \in F$. The language accepted by A, denoted by $\mathcal{L}(A)$, is the subset of Σ^* defined by $\{\pi_\Sigma(r) \mid r \textit{ is an accepting run in } A\}$. Note that $\mathcal{L}(A) = \mathcal{L}(\text{nfa}(A))$.

To remove ε-loops from a pNFA A we define a pNFA flat(A) in Definition 4.6, which has matching time similar to that of A, but contains no ε-loops. The pNFA flat(A) is obtained by replacing consecutive ε-transitions by a single ε-transition without influencing the order on paths.

Definition 4.4 shows how to obtain maximum length subsequences of ε-transitions by using the δ_2-paths of A. If we replace each of these δ_2-paths with a single ε-transition of an appropriate priority, we obtain flat(A). Definition 4.5 can be used to determine appropriate priorities for the new ε-transitions in flat(A). Figure 4.3 shows the result of performing the flattening procedure on a pNFA A recognising the language $(\mathtt{a}^*)^*$. To aid in the understanding of how flat(A) is obtained from A, Table 4.1 shows the corresponding ε-transition in flat(A) for every δ_2-path in A. We extend the definition of flattening to NFAs by proceeding in the same way as for pNFAs, but disregarding priorities at each step.

Definition 4.4 ([16])**.** For a pNFA A, let $r_A(Q_2)$ be the subset of Q_2 defined by $Q_2 \cap (\{q_0\} \cup \{\delta_1(q, \alpha) \mid q \in Q_1, \alpha \in \Sigma\})$, i.e., all Q_2 states reachable from a Q_1 state in one transition, and possibly also q_0. A sequence $p_1 j_2 p_2 \ldots p_{n-1} j_n p_n$, where $p_1 \in r_A(Q_2), p_2, \ldots, p_{n-1} \in Q_2, p_n \in Q_1, j_i \in \mathbb{N}$, is a *$\delta_2$-path* if $\delta_2(p_i) = \ldots p_{i+1} \ldots$, $\delta_2(p_i)$ has at least j_{i+1} occurrences of p_{i+1}, and $(p_i, j_{i+1}, p_{i+1}) = (p_k, j_{k+1}, p_{k+1})$ only if $i = k$. Thus δ_2-paths are maximum length subsequences of distinct ε-transitions (that is, no ε-loops), obtained from paths in a pNFA. We say p_1 is the *initial state* of the δ_2-path.

Definition 4.5 ([16])**.** We define the priority of a δ_2-path relative to the other δ_2-paths with the same initial state as follows. For δ_2-paths $P := p_1 j_2 \ldots j_n p_n$ and let $P' = p'_1 j'_2 \ldots j'_m p'_m$, where $p_1 = p'_1$, we define $P > P'$ (that is, P

has a higher priority than P') if the least i such that $j_i p_i \neq j'_i p'_i$ is such that $\delta_2(p_{i-1}) = \dots p_i \dots p'_i \dots$ where $p_i \neq p'_i$, or $p_i = p'_i$ but $j_i < j'_i$.

Definition 4.6 ([16])**.** We let flat(A) be $(Q_1, r_T(Q_2), \Sigma, q_0, \delta_1, \delta'_2, F)$, where δ'_2 is defined as follows. For $q \in Q'_2$, let $P_1, \dots, P_n$ be all δ_2-paths, ordered according to priority, with initial state q and ending at a state in Q_1. Then $\delta'_2(q) := q_1 \dots q_n$, where q_i is the last state in P_i.

It can be argued that for any input string, matching with A and flat(A) will have similar matching times. Notice that each δ_2-path in A is compressed to a single ε-transition in flat(A), and thus backtracking matching with flat(A) will traverse ε-transitions a constant factor number of times less than when matching with A.

4.4.2 Modeling Backtracking Matching

The backtracking matching process can be modeled by constructing the implicit depth-first search tree which is traversed when matching with the pNFA. This can be achieved via the use of *string-to-tree transducers*. A string-to-tree transducer is a finite state machine which takes a string as input and produces a tree as output. We define a string-to-tree transducer for the pNFA associated with a regex such that given an input string, the output (of the transducer) is the depth-first search tree traversed when using the regex to match the corresponding input string.

Before defining string-to-tree transducers, we first define some related required concepts. A *ranked alphabet* Σ is the union $\Sigma^{(0)} \cup \Sigma^{(1)} \cup \Sigma^{(2)} \dots$ of alphabets $\Sigma^{(i)}$, with only a finite number of the $\Sigma^{(i)}$ nonempty. For a symbol $f \in \Sigma^{(k)}$, we say f has rank k. We allow symbols to have more than one rank, that is $\Sigma^{(0)}, \Sigma^{(1)}, \dots$ may not necessarily be pairwise disjoint. We emphasise the rank of symbol f by writing $f^{(k)}$, when f has rank k. Trees are also ranked, i.e., for a tree t where $t : V \to \Sigma$, where $V \subseteq \mathbb{N}^*$, we have that if $v \in V$ then $vi \in V$, for all $1 \leq i \leq k$, where k is one of the possible ranks of $t(v)$. Recall that $|t| = |V|$ denotes the size of t. Also, let $|t|_S = |\{v \in V \mid t(v) \in S\}|$, for $S \subseteq \Sigma$, be the number of occurrences of symbols from S in t. For notational convenience we write trees where $t(v) \in \Sigma^{(0)} \cup \Sigma^{(1)}$ for all $v \in V$, as $\alpha_1 \alpha_2 \dots \alpha_n$, where $\alpha_i \in \Sigma^{(1)}$ for $i < n$ and $\alpha_n \in \Sigma^{(0)}$. Given a ranked alphabet Δ, the set of all ranked trees $t \colon t_D \to \Delta$ is denoted by T_Δ. Moreover, if Q is an alphabet disjoint from Δ, we let $T_\Delta(Q) := T_{\Delta \cup Q}$ where $Q = Q^{(0)}$, i.e., the symbols in Q appear only at the leaves.

Next we formally define string-to-tree transducers.

Definition 4.7. A *string-to-tree transducer* is a 5-tuple $td = (Q, \Gamma, \Delta, I, \delta)$, where

1. $\Gamma = \Gamma^{(0)} \cup \Gamma^{(1)}$ and Δ are finite ranked input and output alphabets respectively,

2. Q is a finite set of states disjoint from Δ,
3. $I \subseteq Q$ the initial states, and
4. $\delta \subseteq (Q \times \Gamma^{(0)} \times T_\Delta) \cup (Q \times \Gamma_\varepsilon^{(1)} \times T_\Delta(Q))$ is the transition relation.

Also, $|td|_\delta := \sum_{(q,\alpha,t)\in\delta} |t|$ is the *transition size* of td.

When $(q, \alpha_1, t), (q, \alpha_2, t), \ldots, (q, \alpha_n, t) \in \delta$, we write $q \xrightarrow{\alpha_1,\alpha_2,\ldots,\alpha_n} t$; for $(q, \varepsilon, t) \in \delta$, we write $q \to t$; for rules such as $q \xrightarrow{\alpha} t_1$, $q \xrightarrow{\alpha} t_2$, $q \xrightarrow{\alpha} t_3$, we write $q \xrightarrow{\alpha} t_1 \mid t_2 \mid t_3$, and lastly, for $q \to t_1$, $q \to t_2, q \to t_3$, we write $q \to t_1 \mid t_2 \mid t_3$.

For $w \in T_\Gamma$, the set of output trees, when applying td to w, is denoted by $td(w) \subseteq T_\Delta$, and defined as follows. We have that $t \in td(w)$ if w can be written as $\alpha_1 \ldots \alpha_n$, where $\alpha_i \in \Gamma_\varepsilon^{(1)}$ for $i < n$ and $\alpha_n \in \Gamma^{(0)}$, such that there exists a sequence of trees $t_0, \ldots, t_n \in T_\Delta(Q)$ where $t_0 \in I$ and $t_n = t$; and for every $i \in \{1, \ldots, n\}$, t_i is obtained from t_{i-1} by replacing every leaf v for which $t_{i-1}(v) \in Q$ with any tree t' such that $t_{i-1}(v) \xrightarrow{\alpha_i} t'$.

A *backtracking transducer* is a string-to-tree transducer constructed for a pNFA in such a way that given an input string, an output tree consisting of the states traversed during the matching process is returned. Given an input string that is accepted by the pNFA, this is achieved by creating an output tree in which the rightmost branch corresponds to the accepting path in the pNFA. Independent of if the input string is accepted or not, the branches (in the output tree) to the left of the rightmost branch, correspond to higher priority paths in the pNFA which were traversed and backtracked from. For convenience, we assume all input strings have a \$ as end of string marker, and that \$ is not contained within the alphabet over which strings are defined.

A backtracking transducer incorporates the ε-transitions that have been traversed subsequent to the previous symbol transition in its states. In doing this, it is possible to avoid ε-loops by disallowing the traversal of an ε-transition if it is already contained within the state. To simplify the incorporation of ε-transitions into the states, we assume there are no parallel transitions present in the pNFA. This allows us to represent a traversed ε-transition with a tuple (p, q) where p and q are the source and target state of the transition, respectively.

Example 4.1. To illustrate the concept of incorporating ε-transitions into the states of a backtracking transducer, we give the backtracking transducer associated with the pNFA A, recognising the language $\mathcal{L}(E)$ with $E = (\mathtt{a}^*)^*$, shown in Figure 4.3(a). We have that $td_A = (Q, \Gamma, \Delta, \{a_{(q_0,\emptyset)}, f_{(q_0,\emptyset)}\}, \delta)$, where $Q = \{a_{(q,r)}, f_{(q,r)} \mid q \in Q', r \in \mathcal{P}(Q' \times Q')\}$, where $Q' = \{q_0, q_1, q_2, q_3\}$ being the set of states of the pNFA A, $\Gamma = \{\mathtt{a}, \mathtt{b}, \$\}$ and $\Delta = Q' \cup \{\diamond\} = \{q_0, q_1, q_2, q_3, \diamond\}$. The transition rules are:

$$a_{(q_0,\emptyset)} \to q_0[a_{(q_1,\{(q_0,q_1)\})}] \mid q_0[f_{(q_1,\{(q_0,q_1)\})}, a_{(q_3,\{(q_0,q_3)\})}],$$

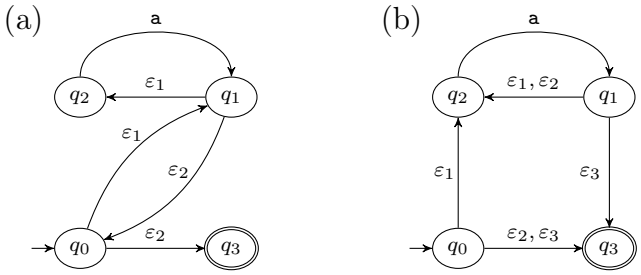

Figure 4.3: (a) A pNFA A recognising the language $\mathcal{L}(E)$ where $E = (\mathtt{a}^*)^*$ and (b) flat(A).

$f_{(q_0,\emptyset)} \to q_0[f_{(q_1,\{(q_0,q_1)\})}, f_{(q_3,\{(q_0,q_3)\})}]$,
$a_{(q_1,\emptyset)} \to q_1[a_{(q_2,\{(q_1,q_2)\})}] \mid q_1[f_{(q_2,\{(q_1,q_2)\})}, a_{(q_0,\{(q_1,q_0)\})}]$,
$f_{(q_1,\emptyset)} \to q_1[f_{(q_2,\{(q_1,q_2)\})}, f_{(q_0,\{(q_1,q_0)\})}]$,
$a_{(q_0,\{(q_1,q_0)\})} \to q_0[a_{(q_1,\{(q_1,q_0),(q_0,q_1)\})}] \mid q_0[f_{(q_1,\{(q_1,q_0),(q_0,q_1)\})}, a_{(q_3,\{(q_1,q_0),(q_0,q_3)\})}]$,
$f_{(q_0,\{(q_1,q_0)\})} \to q_0[f_{(q_1,\{(q_1,q_0),(q_0,q_1)\})}, f_{(q_3,\{(q_1,q_0),(q_0,q_3)\})}]$,
$a_{(q_1,\{(q_0,q_1)\})} \to q_1[a_{(q_2,\{(q_0,q_1),(q_1,q_2)\})}] \mid q_1[f_{(q_2,\{(q_0,q_1),(q_1,q_2)\})}, a_{(q_0,\{(q_0,q_1),(q_1,q_0)\})}]$,
$f_{(q_1,\{(q_0,q_1)\})} \to q_1[f_{(q_2,\{(q_0,q_1),(q_1,q_2)\})}, f_{(q_0,\{(q_0,q_1),(q_1,q_0)\})}]$,
$a_{(q_0,\{(q_0,q_1),(q_1,q_0)\})} \to q_0[a_{(q_3,\{(q_0,q_1),(q_1,q_0),(q_0,q_3)\})}]$,
$f_{(q_0,\{(q_0,q_1),(q_1,q_0)\})} \to q_0[f_{(q_3,\{(q_0,q_1),(q_1,q_0),(q_0,q_3)\})}]$,
$a_{(q_1,\{(q_1,q_0),(q_0,q_1)\})} \to q_1[a_{(q_2,\{(q_1,q_0),(q_0,q_1),(q_1,q_2)\})}]$,
$f_{(q_1,\{(q_1,q_0),(q_0,q_1)\})} \to q_1[f_{(q_2,\{(q_1,q_0),(q_0,q_1),(q_1,q_2)\})}]$,
$a_{(q_2,\{(q_1,q_2)\})} \xrightarrow{\mathtt{a}} q_2[a_{(q_1,\emptyset)}]$,
$f_{(q_2,\{(q_1,q_2)\})} \xrightarrow{\mathtt{a}} q_2[f_{(q_1,\emptyset)}]$,
$f_{(q_2,\{(q_1,q_2)\})} \xrightarrow{\mathtt{b},\$} q_2[\diamond]$,
$a_{(q_2,\{(q_0,q_1),(q_1,q_2)\})} \xrightarrow{\mathtt{a}} q_2[a_{(q_1,\emptyset)}]$,
$f_{(q_2,\{(q_0,q_1),(q_1,q_2)\})} \xrightarrow{\mathtt{a}} q_2[f_{(q_1,\emptyset)}]$,
$f_{(q_2,\{(q_1,q_2),(q_1,q_2)\})} \xrightarrow{\mathtt{b},\$} q_2[\diamond]$,
$a_{(q_2,\{(q_0,q_1),(q_1,q_2)\})} \xrightarrow{\mathtt{a}} q_2[a_{(q_1,\emptyset)}]$,
$f_{(q_2,\{(q_0,q_1),(q_1,q_2)\})} \xrightarrow{\mathtt{a}} q_2[f_{(q_1,\emptyset)}]$,
$f_{(q_2,\{(q_1,q_0),(q_0,q_1),(q_1,q_2)\})} \xrightarrow{\mathtt{b},\$} q_2[\diamond]$,
$a_{(q_3,\{(q_0,q_3)\})} \xrightarrow{\$} q_3[\diamond]$,
$a_{(q_3,\{(q_1,q_0),(q_0,q_3)\})} \xrightarrow{\$} q_3[\diamond]$,
$a_{(q_3,\{(q_0,q_1),(q_1,q_0),(q_0,q_3)\})} \xrightarrow{\$} q_3[\diamond]$.

The intuition behind backtracking transducers is that states $a_{(q,r)}$ and $f_{(q,r)}$ are used to model accepting and rejecting paths respectively. More specifically, transitions originating from states $a_{(q,r)}$ produce subtrees with the rightmost branch corresponding to an accepting path, while transitions from $f_{(q,r)}$ produce subtrees corresponding to paths which are all rejecting. Specifically, consider the rules $a_{(q_0,\emptyset)} \to q_0[a_{(q_1,\{(q_0,q_1)\})}]$ and $a_{(q_0,\emptyset)} \to q_0[f_{(q_1,\{(q_0,q_1)\})}, a_{(q_3,\{(q_0,q_3)\})}]$. The rule $a_{(q_0,\emptyset)} \to q_0[a_{(q_1,\{(q_0,q_1)\})}]$ represents the case in which there is an accepting path starting with a transition from q_0 to q_1. Conversely, the rule $a_{(q_0,\emptyset)} \to q_0[f_{(q_1,\{(q_0,q_1)\})}a_{(q_3,\{(q_0,q_3)\})}]$ represents the case in which the matcher failed to find a match by following the q_0 to q_1 transition (the higher priority transition), but eventually backtracked and accepted the input string by rather following the q_0 to q_3 transition.

Having to account for ε-loops in the states of a backtracking transducer can be very burdensome, resulting in state machines with potentially exponentially many more states and transitions than the corresponding pNFA. A possible workaround is to construct a backtracking transducer rather for flat(A) if A is a pNFA with ε-loops. By definition, flat(A) does not contain any ε-loops and therefore, a less convoluted backtracking transducer can be constructed therefor. Note, since we no longer keep track of ε-transitions in the states of the transducer, we do not require the assumption that we only consider pNFA without parallel transitions.

Example 4.2. As an example of a backtracking transducer for a pNFA without ε-loops, we now give the backtracking transducer for the pNFA flat(A) shown in Figure 4.3(b). We have that $td_A = (Q, \Gamma, \Delta, \{a_{q_0}, f_{q_0}\}, \delta)$, where $Q = \{a_q, f_q \mid q \in \{q_0, q_1, q_2, q_3\}\}$, $\Gamma = \{\mathtt{a}, \mathtt{b}, \$\}$ and $\Delta = \{q_i \mid i \in \{0, 1, 2, 3\}\} \cup \{\diamond\}$. The transition rules are:

$$a_{q_0} \to q_0[a_{q_2}] \mid q_0[f_{q_2}, a_{q_3}] \mid q_0[f_{q_2}, f_{q_3}, a_{q_3}], f_{q_0} \to q_0[f_{q_2}, f_{q_3}, f_{q_3}],$$
$$a_{q_1} \to q_1[a_{q_2}] \mid q_1[f_{q_2}, a_{q_2}] \mid q_1[f_{q_2}, f_{q_2}, a_{q_3}], f_{q_1} \to q_1[f_{q_2}, f_{q_2}, f_{q_3}],$$
$$a_{q_2} \xrightarrow{\mathtt{a}} q_1[a_{q_1}],\ f_{q_2} \xrightarrow{\mathtt{a}} q_1[f_{q_1}],\ f_{q_2} \xrightarrow{\mathtt{b},\$} q_1[\diamond],$$
$$a_{q_3} \xrightarrow{\$} q_3[\diamond].$$

A more intuitive representation of the transition rules for td_A is shown in Figure 4.4. In the figure we use the notation $td_q(w)$ to denote the transducer constructed for a pNFA similar to td, with the only difference being that the initial state for the pNFA corresponding to $td_q(w)$, is state q. The backtracking tree produced when using the input strings `ab` and `aab` is shown in Figures 4.5 and 4.6 respectively.

Example 4.3. As another example of a backtracking transducer for a pNFA without ε-loops, we give the backtracking transducer for the pNFA A shown in Figure 4.2. We have that $td_A = (Q, \Gamma, \Delta, \{a_{q_1}, f_{q_1}\}, \delta)$, where

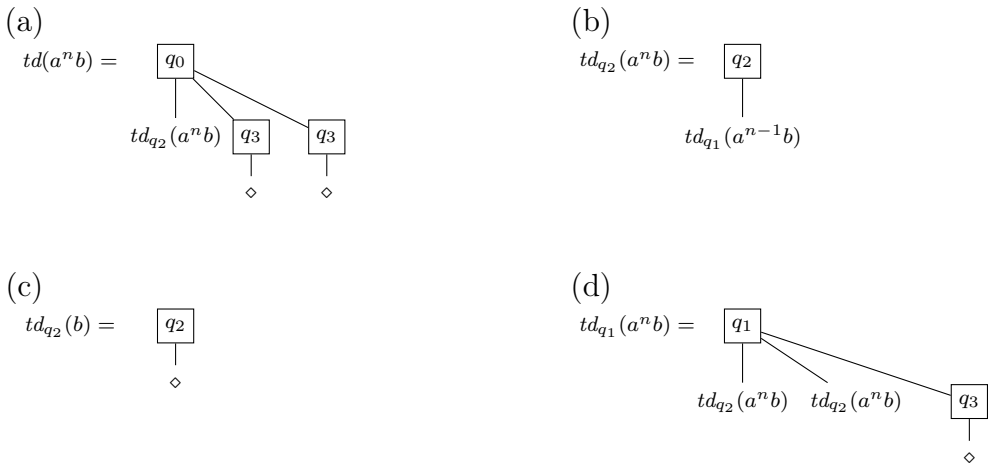

Figure 4.4: The backtracking transducer for the pNFA shown in Figure 4.3(b), in terms of transducers td_{q_1} and td_{q_2}, for input of the form $\mathtt{a}^n\mathtt{b}$.

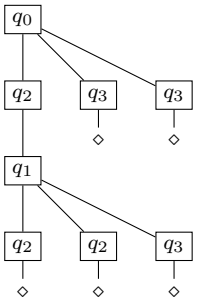

Figure 4.5: The backtracking tree produced when giving the input string $\mathtt{ab}$ to the backtracking transducer of Figure 4.4.

$Q = \{a_q, f_q \mid q \in \{q_0, q_1, q_2, \ldots, q_7\}\}$, $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \$\}$ and $\Delta = \{q_i \mid i \in \mathbb{N}, 0 \leq i \leq 7\} \cup \{\diamond\}$. The transition rules are:

$$a_{q_1} \xrightarrow{\mathtt{a}} q_1[a_{q_2}],\ f_{q_1} \xrightarrow{\mathtt{a}} q_1[f_{q_2}],\ f_{q_1} \xrightarrow{\mathtt{b,c,d,\$}} q_1[\diamond],$$

$$a_{q_2} \to q_2[a_{q_3}] \mid q_2[f_{q_3}, a_{q_4}],\ f_{q_2} \to q_2[f_{q_3}, f_{q_4}],$$

$$a_{q_3} \to q_3[a_{q_4}] \mid q_3[f_{q_4}, a_{q_5}],\ f_{q_3} \to q_3[f_{q_4}, f_{q_5}],$$

$$a_{q_4} \xrightarrow{\mathtt{b}} q_4[a_{q_2}],\ f_{q_4} \xrightarrow{\mathtt{a}} q_4[f_{q_2}],\ f_{q_4} \xrightarrow{\mathtt{b,c,d,\$}} q_4[\diamond],$$

$$a_{q_5} \xrightarrow{\mathtt{b}} q_5[a_{q_2}],\ f_{q_5} \xrightarrow{\mathtt{a}} q_5[f_{q_2}],\ f_{q_5} \xrightarrow{\mathtt{b,c,d,\$}} q_5[\diamond],$$

$$a_{q_6} \xrightarrow{c} q_6[a_{q_2}],\ f_{q_6} \xrightarrow{\mathtt{a}} q_6[f_{q_2}],\ f_{q_6} \xrightarrow{\mathtt{b,c,d,\$}} q_6[\diamond],$$

$$a_{q_7} \xrightarrow{\$} q_7[\diamond].$$

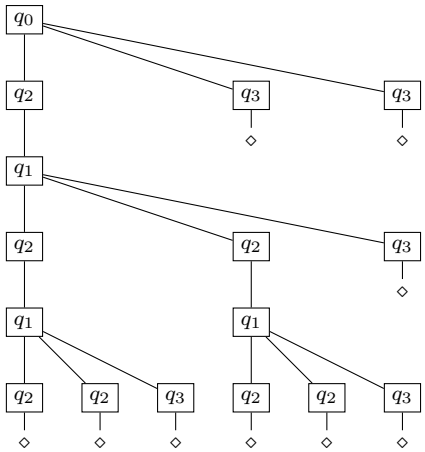

Figure 4.6: The backtracking tree produced when giving the input string `aab` to the backtracking transducer of Figure 4.4.

As with Example 4.2 we provide a more intuitive representation of the transition rules for td_A for input strings of the form $\texttt{ab}^n\texttt{d}$ and $\texttt{ab}^n\texttt{c}$ are shown in Figures 4.7 and 4.8 respectively, and the backtracking trees produced when using input strings `abc` and `abd` are shown in Figures 4.9 and 4.10.

Finally, let us define the backtracking transducer td_A associated with a pNFA A formally. To simplify the formal definition we assume A has no ε-loops (if A has ε-loops, we replace it with $\text{flat}(A)$ instead).

Definition 4.8. For a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, the *backtracking transducer* $td_A = (Q, \Gamma, \Delta, \{a_{q_0}, f_{q_0}\}, \delta)$ is defined as follows. $Q = \{a_q, f_q \mid q \in Q_1 \cup Q_2\}$, $\Gamma^{(0)} = \{\$\}$, $\Gamma^{(1)} = \Sigma$ (we assume $\$ \notin \Sigma$), and $\Delta = Q_1 \cup Q_2 \cup \{\diamond^{(0)}\}$. Furthermore, δ consists of the following transitions:

1. For $q \in Q_1$ and $\alpha \in \Sigma$:
   a) If $\delta_1(q, \alpha) = q'$ let $a_q \xrightarrow{\alpha} q[a_{q'}]$ and $f_q \xrightarrow{\alpha} q[f_{q'}]$.
   b) If $\delta_1(q, \alpha)$ is undefined, $f_q \xrightarrow{\alpha} q[\diamond]$.
2. For $q \in Q_2$, if $\delta_2(q) = q_1 \dots q_n$, then $a_q \to q[f_{q_1}, \dots, f_{q_i}, a_{q_{i+1}}]$ where $0 \leq i \leq n-1$, and $f_q \to q[f_{q_1}, \dots, f_{q_n}]$.
3. Finally, if $q \in F$, $a_q \xrightarrow{\$} q[\diamond]$, whereas when $q \in Q_1 \setminus F$, let $f_q \xrightarrow{\$} q[\diamond]$.

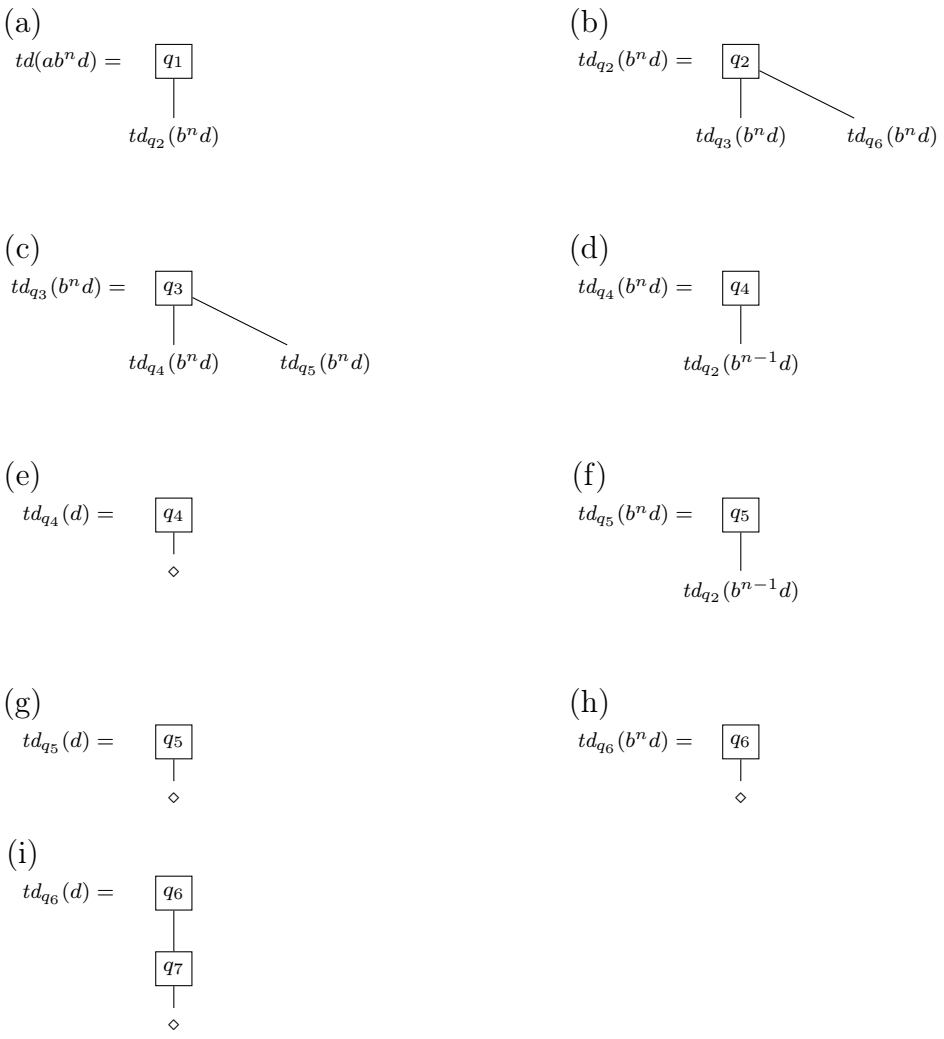

Figure 4.7: The backtracking transducer for the pNFA shown in Figure 4.2, in terms of transducers td_{q_2}, td_{q_3}, td_{q_4}, td_{q_5} and td_{q_6}, for (rejected) input strings of the form $\texttt{a}\texttt{b}^n\texttt{d}$. Note that in (a), (b), (c) and (h) we assume $n \geq 0$, while in (d) and (f) we assume $n \geq 1$.

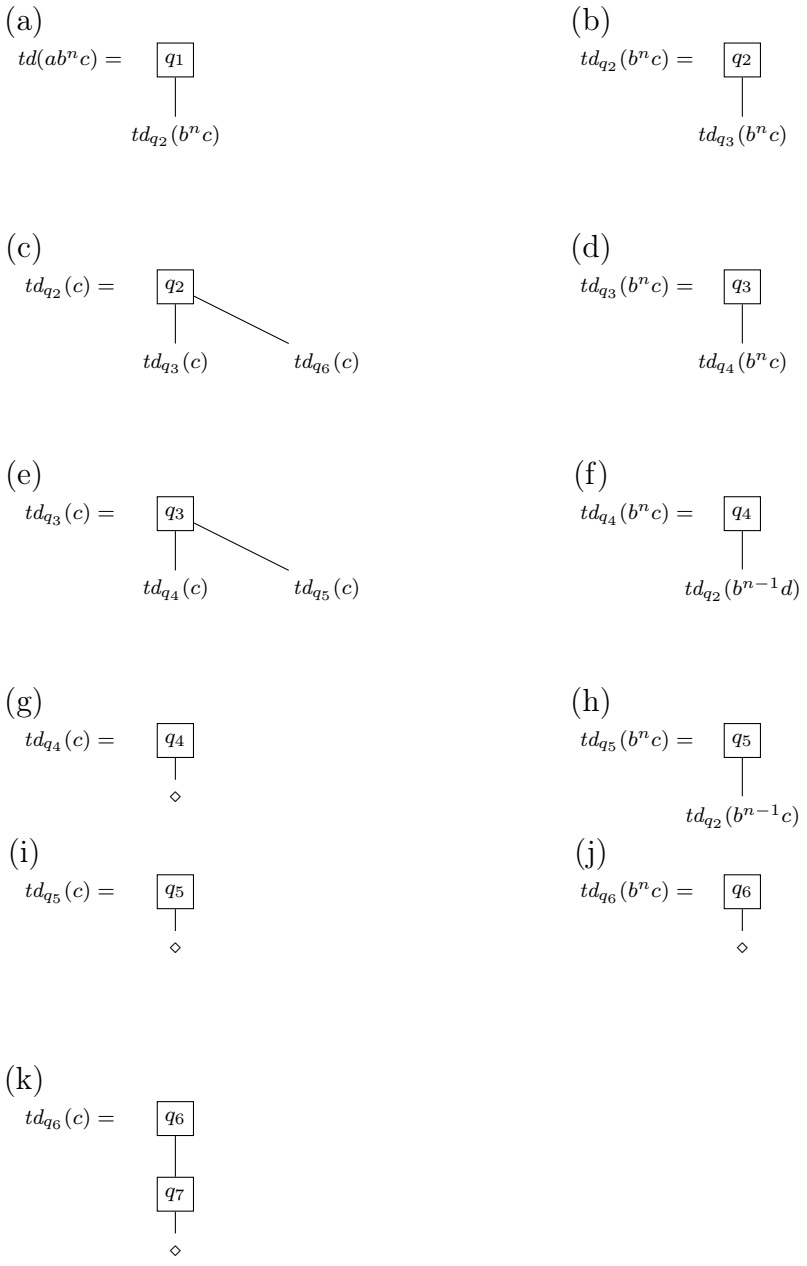

Figure 4.8: The backtracking transducer for the pNFA shown in Figure 4.2, in terms of transducers td_{q_2}, td_{q_3}, td_{q_4}, td_{q_5} and td_{q_6}, for (accepted) input strings of the form $\texttt{ab}^n\texttt{c}$.

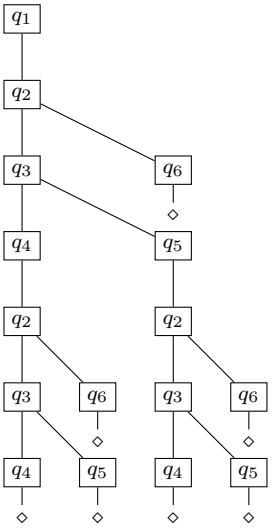

Figure 4.9: The backtracking tree produced when giving the input string `abd` to the backtracking transducer of Figure 4.7.

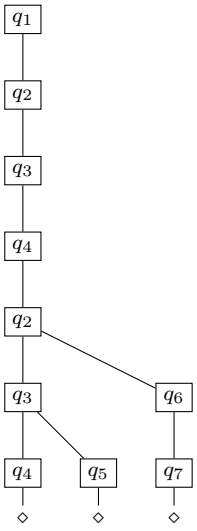

Figure 4.10: The backtracking tree produced when giving the input string `abc` to the backtracking transducer of Figure 4.8.

4.5 Conclusion

This chapter introduced two common matching techniques used by regex engines. We discussed the main differences between subset construction matching and backtracking matching. We also explained how a backtracking matcher can be modeled using pNFAs and backtracking transducers.

Chapter 5

Context and Related Work

5.1 Introduction

In this chapter we discuss the practical significance of vulnerable regexes. We present the context of ReDoS in the field of information security and point out the rather limited amount of related research.

5.2 Algorithmic Complexity Attacks

An algorithmic complexity attack is an exploit on some target algorithm, aiming to trigger the worst-case execution time. If the worst-case execution time is significantly slower than the average case, the attack could cause a state of denial of service in the incorporating program. An algorithmic complexity attack is a type of denial of service attack known as a *sophisticated denial of service attack*. The difference between the standard denial of service attack and its sophisticated counterpart, is that for the sophisticated case, the attacker does not have to send a vast amount of network traffic in order to launch a successful attack [17]. This is usually achieved by exploiting a vulnerability, rather than simply overpowering the victim with superior computational power.

Algorithmic complexity attacks on hash tables provide a well-known class of examples. A hash table guarantees constant lookup and insertion time on average, which is achieved by mapping an object to be inserted, with a *hash function*, to an index in the table. If a malicious user knows the specific hash function being used, it might be possible to cause multiple collisions in the hash table (i.e., having many insertions at the same index). This will cause the insertion and lookup time to become linear, instead of being constant and could potentially slow down the normal functionality of the software under consideration. For a more in-depth discussion of algorithmic complexity attacks on hash tables, refer to [18].

The algorithmic complexity attack relevant to our research is an exploit of the matching algorithm used in backtracking regular expression matchers.

This attack is known as ReDoS.

5.3 Regular Expression Denial of Service

ReDoS is an algorithmic complexity attack on backtracking regular expression matchers. The main idea is to submit a specially crafted input string to be matched by a vulnerable regex, which will force the matcher to cause a denial of service. This is usually achieved by exploiting a highly ambiguous regex in such a way that the matcher is forced to try a vast number of possible ways to match a specific input string. To achieve this, a substring is constructed that can be matched in many possible ways by a subexpression of the regex, due to the ambiguity present in the regex. This substring is then used as part of a larger input string, where the input string is constructed in such a way to force the matcher to try all possible ways of matching the substring.

ReDoS is of practical importance due to the widespread use of regexes. For example, many intrusion detection systems (IDS) use regexes to analyse the contents of data packets entering a network. If one of these regexes are vulnerable, ReDoS could potentially be used to disable the IDS, leaving the network open to other attacks, as shown by [19].

Other than denial of service, it can also be possible in some circumstances to exploit vulnerable regexes to expose sensitive data. This is achieved by using the slow matching time of vulnerable regexes as the base of a timing attack. We discuss a possible timing attack such as this in Appendix B.

5.4 Related Work

Performing static analysis on regexes to estimate the worst-case matching time is a problem that has not yet received much attention in the scientific literature. The concept of ReDoS is explained broadly in [20] and [21], but they only scratch the surface in terms of theory before taking a more practical view. Some examples and experimental results concerning slow matching time are shown in [22].

Our approach is built on ideas from [23], where it is described how to use pNFAs (in theory) to identify regexes with exponential matching time [23]. The authors show how pNFAs can be constructed from regexes to simulate the regex matching algorithm found in Java. Furthermore, they explain how the method of pNFA construction used, affects the input strings for which an evil regex is vulnerable.

The authors in [24] approach the problem of identifying vulnerable regexes by creating an abstract syntax tree (AST) from a regex, which is then used to construct an equivalent NFA. For this NFA an abstract machine is constructed, that allows for the modeling of backtracking regex matchers. The static anal-

ysis then proceeds to search over this abstract machine for paths indicating exponential matching time. This analysis, however, does not address some of the more in-depth issues related to estimating worst-case matching time of backtracking matchers. One such issue is how the fact that certain backtracking matchers prioritise some matches over others and cease the matching process as soon as a match has been found, affects the vulnerability of a regex. These complications introduced by prioritisation during the matching process, is addressed in [25], by using the concept of an *ordered NFA*. An ordered NFA is very similar to a pNFA. The difference between these two models, is that the ordered NFA does not use ε-transitions, prioritised or otherwise. Instead, an ordered NFA uses ordered multistates to define an order over the destination states of each transition from a state, for an input symbol. Similar to the prioritised ε-transitions of a pNFA, this order prioritises certain matches above others. However, the exclusion of ε-transitions in an ordered NFA eliminates the need to remove ε-loops. They also describe how to find strings that trigger the exponential matching time.

In [26] and [27] a problem known as the *output size problem* for string-to-tree transducers is discussed. The output size problem for string-to-tree transducers is to determine the rate of growth of the largest possible output tree in relation to the length of an input string, for a string-to-tree transducer. It is shown that this problem is related to determining the worst-case matching time of a regex for some backtracking regex matcher. It uses the results on NFA ambiguity from [14] to approximate the output size of a given string-to-tree transducer in certain cases.

Our main contribution to the body of research dealing with ReDoS is the development of static analysis techniques to reveal both high-degree polynomial and exponential matching time.

5.5 Conclusion

This chapter discussed algorithmic complexity attacks in general and also focused on the practical importance of investigating ReDoS.

Chapter 6

Linking Regexes, Ambiguity and Matching Time

6.1 Introduction

In Chapters 2, 3 and 4 we explained the concepts of ambiguity of NFAs, regexes and backtracking matching. In this chapter we describe in more detail the link between all these concepts. This is accomplished by defining two types of regex ambiguity and describing a regex-to-NFA construction which preserves both kinds of ambiguity. We define weak and strong ambiguity as a generalisation of the concepts defined in [28], and [29], respectively. In [30], both concepts are discussed and contrasted. We also show how to construct pNFAs, from regexes, in order to model the matching time of the Java regex matcher.

6.2 Weak Ambiguity

The term weak ambiguity of a regex relates to how the symbols of an input string can be matched by the corresponding symbols in the regex. A weakly unambiguous regex can match the symbols of an input string with the corresponding symbols in the regex in at most one way. A regex is weakly ambiguous if it is not weakly unambiguous. For an example of a weakly unambiguous regex, consider the regex $E_1 = \mathtt{a}^*$. For every input string $\mathtt{a}^k$, each of the k $\mathtt{a}$ symbols can only be matched by the $\mathtt{a}$ symbol in E_1, and therefore the regex E_1 is weakly unambiguous. Conversely, consider the regex $E_2 = (\mathtt{a} \mid \mathtt{a})$. The input string $\mathtt{a}$ can be matched by any of the two branches in the alternation of E_2, and consequently, E_2 is weakly ambiguous.

To define weak ambiguity formally, we first give Definition 6.1, which is used to annotate regexes. Recall from Chapter 2 that subexpressions in a regex E are numbered in a depth-first preorder. We use $E(i)$ to denote subexpression i of the regex E (e.g. for $E = ((\mathtt{ab}) \mid \mathtt{b})^*$ we have $E(1) = E$, $E(3) = \mathtt{ab}$ and $E(4) = \mathtt{a}$).

Definition 6.1. For a regex E over the alphabet Σ (with $\Sigma \cap \mathbb{N} = \emptyset$) define $Y_{\mathrm{wA}}(E)$ by replacing subexpression $E(i)$ with $i \cdot E(i)$ if $E(i) \in \Sigma$. That is, $Y_{\mathrm{wA}}(E)$ is a regex over $\Sigma \cup \mathbb{N}$.

Using Definition 6.1, we define weak ambiguity next.

Definition 6.2. For a regex E let

$$\mathrm{wA}_E(n) = \max_{w \in \Sigma^*, |w| \leq n} |\{v \in \mathcal{L}(Y_{\mathrm{wA}}(E)) \mid \pi_\Sigma(v) = w\}|.$$

wA_E is the *degree of weak ambiguity* of E. If $\mathrm{wA}_E(n) \leq 1$ for all n, E is *weakly unambiguous.* If there exists a constant c such that $\mathrm{wA}_E(n) \leq c$, E has *constant weak ambiguity.* If $\mathrm{wA}_E(n)$ is a polynomial function in n of degree d, with $d \geq 1$, then E has *infinite degree of weak ambiguity* of degree d. Similarly, if $\mathrm{wA}_E(n)$ is an exponential function in n, then E has *exponential degree of weak ambiguity.*

We illustrate the previous definition with the regex $E_2 = (\mathtt{a} \mid \mathtt{a})$. Note that $Y_{\mathrm{wA}}(E_2) = (2\mathtt{a} \mid 3\mathtt{a})$ and since $Y_{\mathrm{wA}}(E_2)$ matches $w_1 = 2\mathtt{a}$ and $w_2 = 3\mathtt{a}$, E_2 is weakly ambiguous, as $\pi_\Sigma(w_1) = \pi_\Sigma(w_2) = \mathtt{a}$.

To illustrate the concepts of infinite and exponential degree of weak ambiguity, first consider the regex $E_3 = \mathtt{a}^*\mathtt{a}^*$. Then $Y_{\mathrm{wA}}(E_3) = (3\mathtt{a})^*(5\mathtt{a})^*$, and $Y_{\mathrm{wA}}(E_3)$ matches strings of the form $(3\mathtt{a})^{n_0}(5\mathtt{a})^{n_1}$, with $n_0, n_1 \geq 0$. Since $\pi_\Sigma((3\mathtt{a})^{n_0}(5\mathtt{a})^{n_1}) = \mathtt{a}^{n_0+n_1}$, $\mathrm{wA}_{E_3}(n) = n + 1$ and E_3 has weak ambiguity of degree 1.

For the case of exponential degree of ambiguity, consider the regex $E_4 = (\mathtt{a} \mid \mathtt{a})^*$. The annotated regex $Y_{\mathrm{wA}}(E_4) = (3\mathtt{a} \mid 4\mathtt{a})^*$ matches 2^n strings v such that $\pi_\Sigma(v) = \mathtt{a}^n$, and thus E_4 has exponential degree of weak ambiguity.

6.3 Glushkov Construction

Glushkov's construction algorithm [30; 31] operates over a set of positions in a regex, along with functions defined to describe the behaviour of the regex. It constructs an automaton which preserves weak ambiguity of a regex. We denote this automaton, constructed from regex E, as $Gl(E)$.

The set of positions in a regex E is obtained by adding a unique subscript to each subexpression that is a symbol from Σ, forming regex E'. This idea is similar to the concept of annotating regexes as $iE(i)$ for each subexpression $E(i)$, introduced in the previous sections, but where the previous method of annotation changed the language matched by the regex, adding subscripts does not. Formally, $\mathrm{pos}(E) = \{i \mid E(i) \in \Sigma\}$ denotes the set of positions of E. For example, to the regex $E = \mathtt{a}(\mathtt{b} \mid \mathtt{c})^*\mathtt{d}$, subscripts are added as in $E' = \mathtt{a}_2(\mathtt{b}_6 \mid \mathtt{c}_7)^*\mathtt{d}_8$, and $\mathrm{pos}(E) = \{2, 6, 7, 8\}$. Let $\chi_R(x)$ denote the symbol in regex R at position x, for example, $\chi_{E'}(2) = \mathtt{a}$.

To describe the matching behaviour of a regex, the functions *first*, *last* and *follow* are defined. The function first(E') maps the regex E' to the positions of symbols that can match the first symbol of some input string. In our previous example with the regex E', first(E') $= \{2\}$ and for regexes such as $(\mathtt{a}_2 \mid \mathtt{b}_3)$ and $(\mathtt{a}_3{}^*\mathtt{b}_4)$, first($E'$) $= \{2, 3\}$ and first(E') $= \{3, 4\}$, respectively. Similarly, the function last(E') maps the regex E' to the positions at which some input string $w \in \mathcal{L}(E)$ can be accepted. In the case of the regex E', last(E') $= \{8\}$ and for regexes such as $(\mathtt{a}_2 \mid \mathtt{b}_3)$ and $(\mathtt{a}_2\mathtt{b}_4{}^*)$, last($E'$) $= \{2, 3\}$ and last(E') $= \{2, 4\}$, respectively.

The function follow(E', x), where $x \in$ pos(E), describes the behaviour of the regex while symbols from Σ are being read. It maps a position x in the regex to a set of positions that can be reached from position x when reading a symbol from Σ. In our example, follow($E', 2$) $= \{6, 7, 8\}$, follow($E', 6$) $= \{6, 7, 8\}$, follow($E', 7$) $= \{6, 7, 8\}$ and follow($E', 8$) $= \emptyset$.

Obtaining these functions are trivial for simple regexes, but in general, these functions need to be built up recursively, similarly to how an automaton is built recursively from a given regex. This can be achieved using Definitions 6.3 to 6.7.

Definition 6.3 ([30])**.** (Empty string recognising regex, $E' = \varepsilon$)
We define:

1. first(E') $= \emptyset$,
2. last(E') $= \emptyset$, and
3. since, pos(E) $= \emptyset$, the follow function is not defined.

Definition 6.4 ([30])**.** (Single symbol string recognising regex, $E' = \alpha_i$)
We define:

1. first(E') $= \{i\}$,
2. last(E') $= \{i\}$, and
3. follow(E', i) $= \emptyset$.

Definition 6.5 ([30])**.** (Union of two regexes $E' = F' \mid G'$)
Let pos(E) $=$ pos(F)$\cup$pos(G) and we assume pos(F)$\cap$pos(G) $= \emptyset$. We define:

1. first(E') $=$ first(F') $\cup$ first(G'),
2. last(E') $=$ last(F') $\cup$ last(G'), and
3. for $x \in$ pos(E),

$$\text{follow}(E', x) = \begin{cases} \text{follow}(F', x) & \text{if } x \in \text{pos}(F); \\ \text{follow}(G', x) & \text{if } x \in \text{pos}(G). \end{cases}$$

Definition 6.6 ([30])**.** (Concatenation of two regexes $E' = F'G'$)
Let $\text{pos}(E) = \text{pos}(F) \cup \text{pos}(G)$ and we assume $\text{pos}(F) \cap \text{pos}(G) = \emptyset$. We define:

1. $\text{first}(E') = \begin{cases} \text{first}(F') \cup \text{first}(G') & \varepsilon \in \mathcal{L}(F); \\ \text{first}(F') & \text{otherwise;} \end{cases}$
2. $\text{last}(E') = \begin{cases} \text{last}(F') \cup \text{last}(G') & \varepsilon \in \mathcal{L}(G); \\ \text{last}(G') & \text{otherwise;} \end{cases}$
3. for $x \in \text{pos}(E)$,

$$\text{follow}(E', x) = \begin{cases} \text{follow}(F', x) & x \in \text{pos}(F) \setminus \text{last}(F'); \\ \text{follow}(F', x) \cup \text{first}(G') & x \in \text{last}(F'); \\ \text{follow}(G', x) & x \in \text{pos}(G). \end{cases}$$

Definition 6.7 ([30])**.** (Kleene star of a regex $E' = F'^*$)
We define:

1. $\text{first}(E') = \text{first}(F')$,
2. $\text{last}(E') = \text{last}(F')$, and
3. for $x \in \text{pos}(E)$,

$$\text{follow}(E', x) = \begin{cases} \text{follow}(F', x) & x \in \text{pos}(F) \setminus \text{last}(F'); \\ \text{follow}(F', x) \cup \text{first}(F') & x \in \text{last}(F'). \end{cases}$$

Using the positions of a regex, along with the definitions of first, last and follow, an automaton can be created using Definition 6.8.

Definition 6.8. Glushkov's Construction
From the positions and definitions of the functions first, last and follow, Glushkov's construction creates an NFA $A = (Q, \Sigma, \delta, q_\text{I}, F)$ from a regex E. Assume E' is regex E with subscripts added from $\text{pos}(E)$:

1. $Q = \{q_i \text{ for } i \in \text{pos}(E)\} \cup \{q_\text{I}\}$, i.e. the states of A are the positions of E with a new state q_I.
2. For $\alpha \in \Sigma$ let $\delta(q_\text{I}, \alpha) = \{q_x | x \in \text{first}(E'), \chi_{E'}(x) = \alpha\}$.
3. For $\alpha \in \Sigma$ and $x \in \text{pos}(E)$, let $\delta(q_x, \alpha) = \{q_y | y \in \text{follow}(E', x), \chi_{E'}(y) = \alpha\}$.
4. $F = \begin{cases} \text{last}(E') \cup \{q_\text{I}\} & \text{if } \varepsilon \in \mathcal{L}(A); \\ \text{last}(E') & \text{otherwise.} \end{cases}$

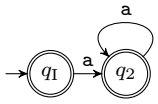

Figure 6.1: The automaton $Gl(E_1)$ for the regex $E_1 = \mathtt{a}^*$.

As an example of Glushkov construction, we construct the NFA $Gl(E_1)$, where $E_1 = \mathtt{a}^*$ and $E_1' = \mathtt{a}_2{}^*$. For E_1, $\text{first}(E_1') = \{2\}$, $\text{last}(E_1') = \{2\}$ and $\text{follow}(E_1', 2) = \{2\}$. The NFA $Gl(E_1)$ is shown in Figure 6.1.

To illustrate via an example that the Glushkov construction preserves weak ambiguity, we consider the weakly ambiguous regex $E_2 = (\mathtt{a} \mid \mathtt{a})^*$. First, we add subscripts to E_2, $E_2' = (\mathtt{a}_3 \mid \mathtt{a}_4)^*$. Then using E_2', we calculate $\text{first}(E_2') = \text{last}(E_2') = \text{follow}(E_2', 3) = \text{follow}(E_2', 4) = \{3, 4\}$. The automaton $Gl(E_2)$ is shown in Figure 6.2. Note that by using the definitions of ambiguity for regexes and ambiguity for automata found in Section 2.7, we see that the regex E_2 has exponential degree of weak ambiguity and $Gl(E_2)$ also has exponential ambiguity.

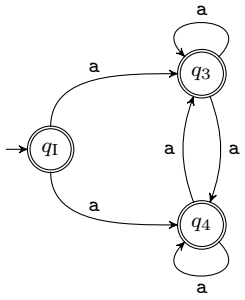

Figure 6.2: The NFA $Gl(E_2)$ for the regex $E_2 = (\mathtt{a} \mid \mathtt{a})^*$.

6.4 Strong Ambiguity

Weak ambiguity of regexes is insufficient to encompass accurately all the intricacies of the ambiguity found in automata obtained from regexes, by using for example Thompson construction (discussed in Section 6.5). Therefore, we define another type of regex ambiguity, namely *strong ambiguity.* Strong ambiguity of a regex relates to how the symbols of an input string are matched with all the subexpressions of the regex. First we define how regexes are annotated for strong ambiguity.

Definition 6.9. For a regex E over the alphabet Σ (with $\Sigma \cap \mathbb{N} = \emptyset$) define the regex $Y_{\mathrm{sA}}(E)$ by replacing subexpression $E(i)$ with $i \cdot E(i)$, for each i. That is, $Y_{\mathrm{sA}}(E)$ is a regex over $\Sigma \cup \{1, \ldots, |E|\}$.

Further, let $L = \bigcup\{(\Sigma \cup \mathbb{N})^* x \mathbb{N}^* x (\Sigma \cup \mathbb{N})^* \mid x \in \mathbb{N}\}$, and define $X_{\mathrm{sA}}(E) = \mathcal{L}(Y_{\mathrm{sA}}E) \cap \overline{L}$, where $\overline{L}$ is all strings not in L.

Intuitively, $X_{\mathrm{sA}}(E)$ is the strings of $\mathcal{L}(E)$ annotated with the subexpressions which can be visited during the matching, *except* disallowing visiting the same subexpression twice without reading a symbol from Σ in between (enforced by the intersection with $\overline{L}$). This mimics the behaviour of Definition 4.3 in terms of disallowing ε-transition repetitions, and hence the backtracking matching behaviour in terms of avoiding infinite loops. Next we define the degree of strong ambiguity of a regex.

Definition 6.10. For a regex E let

$$\mathrm{sA}_E(n) = \max_{w \in \Sigma^*, |w| \leq n} |\{v \in X_{\mathrm{sA}}(E) \mid \pi_\Sigma(v) = w\}|.$$

sA_E is the *degree of strong ambiguity* of E (where the max over the empty set is taken to be 0). If $\mathrm{sA}_E(n) \leq 1$ for all n, E is *strongly unambiguous.* If there exists a constant c such that $\mathrm{sA}_E(n) \leq c$, E has *constant strong ambiguity.* If $\mathrm{sA}_E(n)$ is a polynomial function in n of degree d, then E has *infinite degree of strong ambiguity* of degree d. Similarly, if sA_E is an exponential function, then E has *exponential degree of strong ambiguity.* Note that any strongly unambiguous regex will also be weakly unambiguous.

To illustrate the concept of strong ambiguity, consider the regex $E_5 = (\varepsilon \mid \mathtt{a})^*$. First we create the annotated regex $Y_{\mathrm{sA}}(E_5) = 1(2(3\varepsilon \mid 4\mathtt{a}))^*$, which matches $w_1 = 124\mathtt{a}24\mathtt{a}$ and $w_2 = 124\mathtt{a}24\mathtt{a}23$, and thus since $\pi_\Sigma(w_1) = \pi_\Sigma(w_2) = \mathtt{aa}$, E_5 has strong ambiguity. Note that, by definition of Y_{sA}, a string such as $124\mathtt{a}2324\mathtt{a}$ is not included in $X_{\mathrm{sA}}(E_5)$, due to presence of the substring 232. In general, $X_{\mathrm{sA}}(E_5)$ contains strings of the forms $1(24\mathtt{a})^n$ and $1(24\mathtt{a})^n23$, and and therefore E_5 has constant strong ambiguity.

In contrast, E_5 is weakly unambiguous, since $Y_{\mathrm{wA}}(E_5) = (\varepsilon \mid 4\mathtt{a})^*$ matches only strings of the form $w_n = (4\mathtt{a})^n$. Indeed, the automaton $Gl(E_5)$ is the same as $Gl(E_1)$ given in Section 6.3, with $E_1 = \mathtt{a}^*$. To show this, let us use the subscript 2 for $E_5(4) = \mathtt{a}$ instead of 4, $E_5' = (\varepsilon \mid \mathtt{a}_2)^*$. This yields, for the regex E_5, $\mathrm{first}(E_5') = \mathrm{last}(E_5') = \mathrm{follow}(E_5', 2) = \{2\}$. Since $\mathrm{first}(E_1') = \mathrm{first}(E_5')$, $\mathrm{last}(E_1') = \mathrm{last}(E_5')$ and $\mathrm{follow}(E_1', 2) = \mathrm{follow}(E_5', 2)$, both regexes will have the same Glushkov automaton, and thus the Glushkov construction does not preserve strong ambiguity.

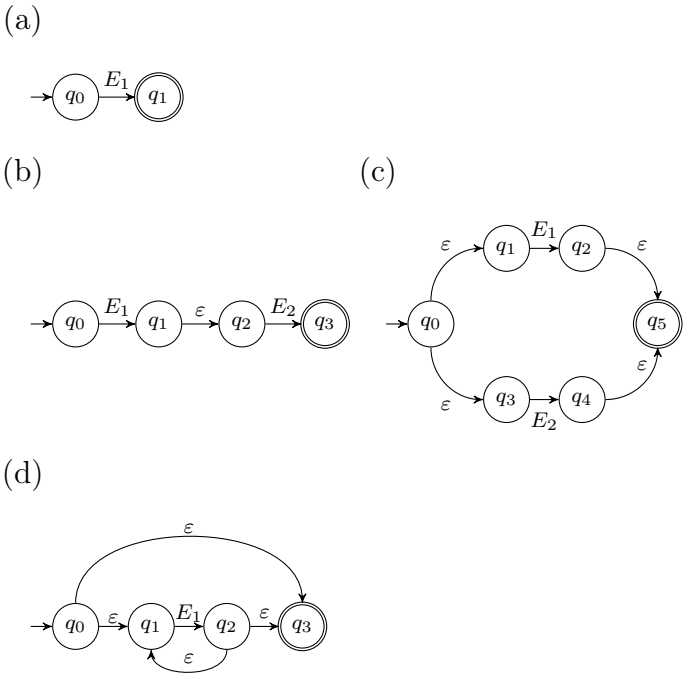

Figure 6.3: Thompson NFAs for (a) $Th(E_1)$, (b) $Th(E_1 \cdot E_2)$, (c) $Th(E_1 \mid E_2)$ and (d) $Th(E_1^*)$.

6.5 Thompson Construction

Another well-known regex-to-NFA construction method is Thompson's method [32]. This method preserves strong ambiguity of regexes, unlike Glushkov's construction. Given a regex E, we use $Th(E)$ to denote the NFA obtained by using Thompson construction. The construction of $Th(E)$, when using concatenation, alternation and the Kleene star operator, is shown in Figure 6.3.

To illustrate the strong ambiguity preserving feature of Thompson construction, we use the regex $E_5 = (\varepsilon \mid \mathtt{a})^*$, which was shown to be strongly ambiguous in Section 6.4. We show $Th(E_5)$ in Figure 6.4(a). The automaton $Th(E_5)$ is ambiguous, as there are multiple paths for reading the input string $\mathtt{a}$. Two of these paths are $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\varepsilon} q_3 \xrightarrow{\varepsilon} q_4 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_6 \xrightarrow{\mathtt{a}} q_7 \xrightarrow{\varepsilon} q_4 \xrightarrow{\varepsilon} q_5$ and $q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_6 \xrightarrow{\mathtt{a}} q_7 \xrightarrow{\varepsilon} q_4 \xrightarrow{\varepsilon} q_5$. If we disallow the repeated usage of ε-transitions without reading any input symbols, this automaton has constant degree of ambiguity. After the first $\mathtt{a}$ symbol has been read, the automaton can only be in state q_1 if the $q_4 \xrightarrow{\varepsilon} q_1$ transition has been traversed. Therefore, the $q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\varepsilon} q_3 \xrightarrow{\varepsilon} q_4 \xrightarrow{\varepsilon} q_1$ loop can only be traversed once at the start of the input string (before the first $\mathtt{a}$ symbol has been read). We can simulate the prevention of repeated ε-transition usage, by removing the ε-loops of $Th(E_5)$. The result, flat($Th(E_5)$), is shown in Figure 6.4(b). As a consequence of re-

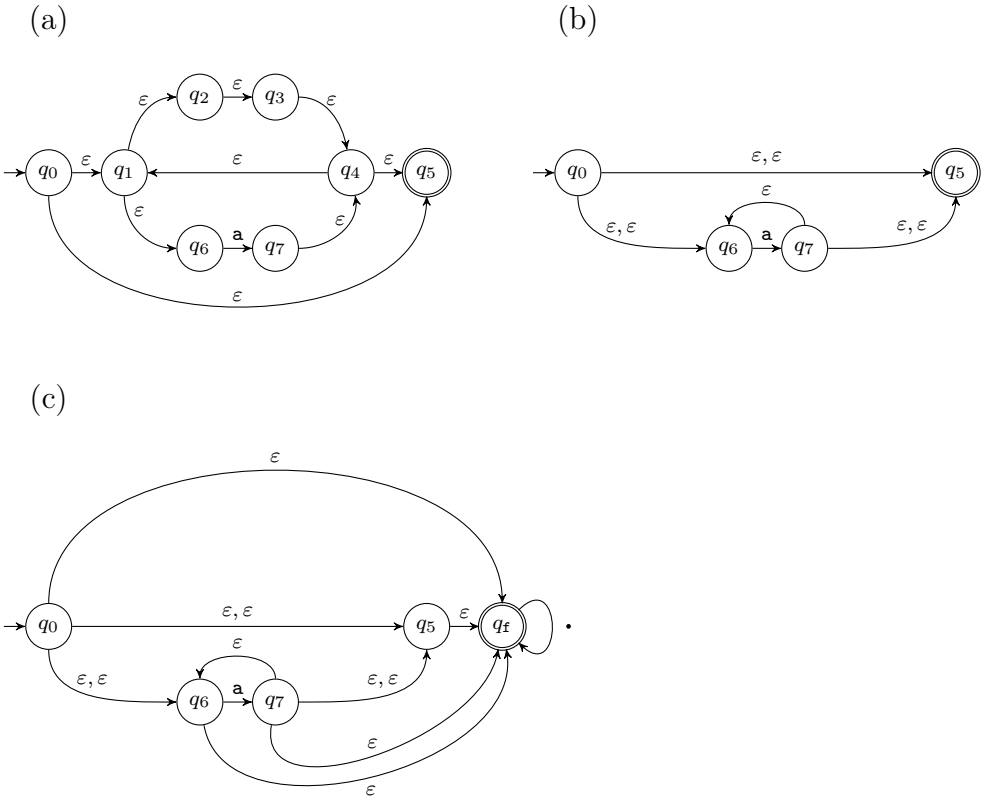

Figure 6.4: The NFAs (a) $Th(E)$ with $E = (\varepsilon \mid \mathtt{a})^*$, (b) flat($Th(E)$) and (c) M'_A with $A = Th(E)$.

moving the ε-loops, the number of paths to read $\mathtt{a}^k$ is reduced to at most 4. Therefore, the NFA has constant degree of ambiguity, similar to E_5, which has constant degree of strong ambiguity.

If ε-loops are removed from NFAs, it is possible to relate the worst-case matching time of backtracking matchers (discussed in Chapter 4) witnessed when matching with E, and the degree of ambiguity of an NFA constructed from E. We illustrate this using Thompson construction. To achieve this, the constructed automaton is slightly adapted. For a regex E and an NFA $A = Th(E)$, let $M_A = \text{flat}(A)$. From M_A, we construct M'_A, such that the degree of ambiguity of M'_A determines an upper bound (often tight) for matching time. If $M_A = (Q, \Sigma, q_0, \delta, F)$, we construct M'_A such that $M'_A = (Q \cup \{q_\mathtt{f}\}, \Sigma, q_0, \delta', \{q_\mathtt{f}\})$, with the transition function δ' defined as:

$$\delta'(p, \alpha, q) = \begin{cases} \delta(p, \alpha, q) & p, q \in Q, \alpha \in \Sigma^\varepsilon; \\ 1 & p \in Q, q = q_\mathtt{f}, \alpha = \varepsilon; \\ 1 & p = q = q_\mathtt{f}, \alpha \neq \varepsilon; \\ 0 & \text{otherwise.} \end{cases}$$

Informally, to construct M'_A, we add a new accept state $q_{\mathtt{f}}$ to M_A (and change all original accept states to be nonaccept states). We then add ε-transitions from all states in M_A to $q_{\mathtt{f}}$ to M_A and a self-loop on $q_{\mathtt{f}}$ for all symbols. The degree of ambiguity of M'_A provides an upper bound for the worst-case matching time of E. To understand why it is necessary to adapt A into M'_A, recall that the backtracking matcher might attempt to match a prefix of an input string in an exponential number of ways, even though the match will ultimately fail. If we can force the matcher to try all possible paths to match (prefixes of) an input string, this upper bound is tight. The tight upper bound is usually achieved when it is possible to force the matcher to reject an input string, after all these paths have been attempted.

We give an example of the process of obtaining an upper bound for the matching time, with the regex $E_5 = (\varepsilon \,|\, \mathtt{a})^*$. The NFAs $Th(E_5)$, flat($Th(E_5)$) and M'_{A_5}, for $A_5 = Th(E_5)$, are shown in Figure 6.4. Observe that the NFA in Figure 6.4(c) has IDA of degree 1. The ambiguity of this NFA provides a tight upper bound to matching with E_5, since E_5 has linear worst-case matching time.

As an example of estimating an upper bound when worst-case matching time is exponential, consider the regex $E_6 = (\mathtt{a}^*)^*$. This regex has exponential degree of strong ambiguity. To show this, we first annotate E_6 as $Y_{\mathrm{sA}}(E_6) = 1(2(3\mathtt{a})^*)^*$. The regex $Y_{\mathrm{sA}}(E_6)$ matches $123\mathtt{a}s_1s_2\ldots s_n$ (in addition to some other strings), where $s_i \in \{3\mathtt{a}, 23\mathtt{a}\}$, for $i \geq 1$. Since there is an exponential, in n, number of strings of the form $w = 123\mathtt{a}s_1s_2\ldots s_n$, and since for all of them $\pi_\Sigma(w) = \mathtt{a}^{n+1}$, we conclude that E_6 has exponential degree of strong ambiguity. We show $A_6 = Th(E_6)$, flat($Th(E_6)$) and M'_{A_6} in Figure 6.5. From these figures, it is clear that M'_{A_6} has EDA and therefore, we can estimate an upper bound of exponential matching time for E_6, which is tight.

6.6 Java Construction

In Chapter 4, we stated that a backtracking matcher will prioritise matches made with certain subexpressions, over others, as well as that a backtracking matcher will cease the matching process as soon as a successful match has been found (or all possible paths have been exhausted). Clearly, this is an indication that priorities can have an effect on matching time. As an example, consider the regex $E_7 = (\mathtt{.}^* \,|\, (\mathtt{a}^*)^*)$. As we have seen in Section 6.5, the subexpression $E_7(4) = (\mathtt{a}^*)^*$ has exponential degree of strong ambiguity, and thus, E_7 has at least exponential degree of strong ambiguity. At the same time, the backtracking matcher prioritises matches made with subexpression $E_7(2) = \mathtt{.}^*$, over those made with $E_7(4)$. Consequently, as $E_7(2)$ will accept any input string and the matcher will stop the matching processes as soon as a string has been accepted, the matcher will accept any string in linear time.

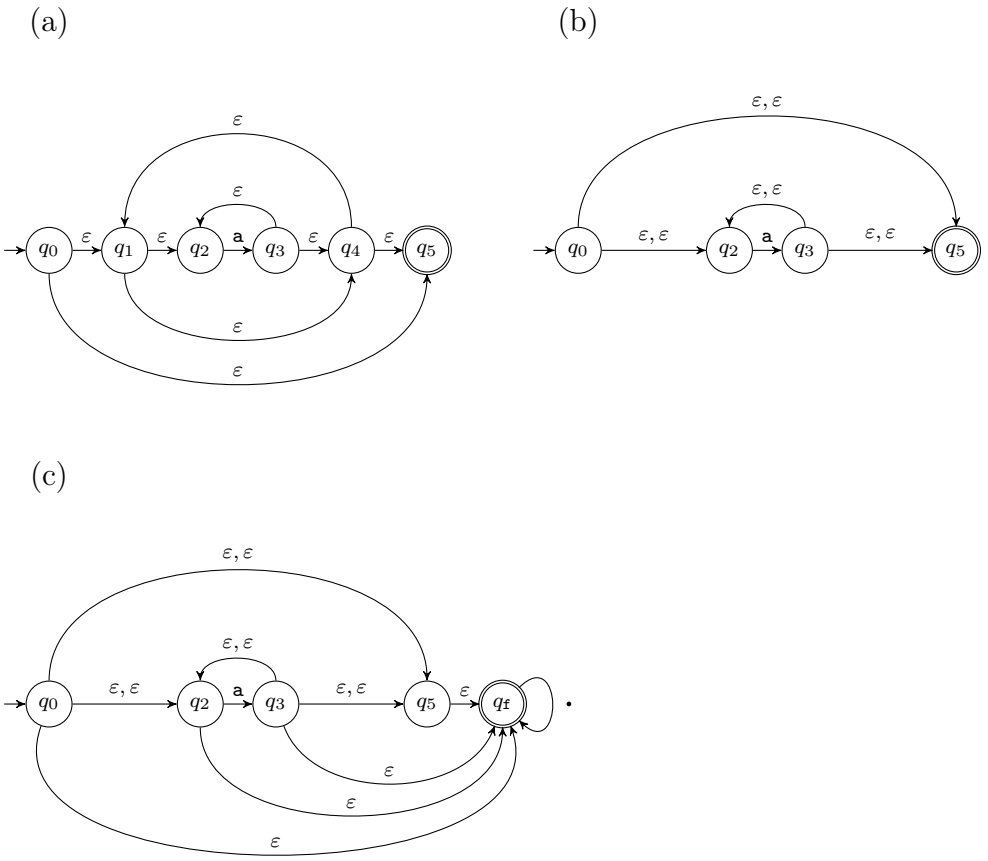

Figure 6.5: The NFAs (a) $Th(E)$ where $E = (\mathtt{a}^*)^*$, (b) flat($Th(E)$) and (c) M'_A where $A = Th(E)$.

To obtain a precise bound on worst-case matching time, we need to incorporate the priorities of how a regex matches an input string, into our backtracking matcher model. This can be achieved by including priorities in the regex to automaton construction, so that it creates pNFAs, instead of standard NFAs. We provide an adapted Thompson construction, which creates pNFAs with priorities placed on the ε-transitions in such a way to model the priorities of a backtracking matcher [23]. We refer to this adapted Thompson construction as the *prioritised Thompson construction*, and denote the pNFA obtained from applying this procedure to a regex E, as $Th^p(E)$. The construction of $Th^p(E)$, when using concatenation, alternation, Kleene star and the lazy Kleene star operator, is shown in Figure 6.6. Even though prioritised Thompson construction is a step in the right direction, unfortunately, it cannot be used to model arbitrary backtracking matchers. Due to the ad hoc implementation of most backtracking matchers, the actual matching time and the (asymptotic) matching time estimated by the model, could differ. Before we illustrate this

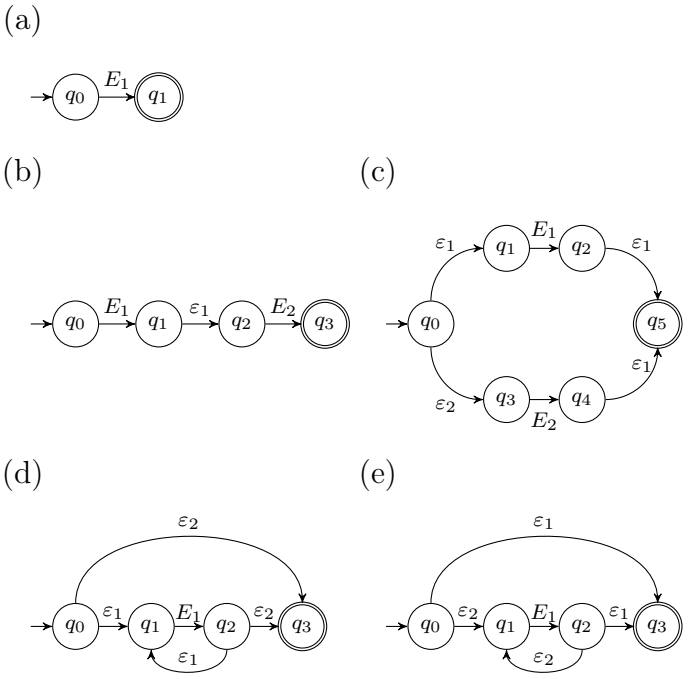

Figure 6.6: Thompson pNFAs for (a) $Th^p(E_1)$, (b) $Th^p(E_1 \cdot E_2)$, (c) $Th^p(E_1 \mid E_2)$, (d) $Th^p(E_1^*)$ and (e) $Th^p(E_1^{*?})$.

with an example, we first define the Java construction from [23], which models the matching time of the Java matcher more accurately than the prioritised Thompson construction.

The *Java construction* for a regex E, denoted by $J^p(E)$ for a regex E, is shown in Figure 6.7 for concatenation, alternation, Kleene star and the lazy Kleene star operator. Using the Java construction, we can begin to understand the long worst-case matching time of some regexes, when using the Java regex matcher. In Figure 6.8, the pNFAs $J^p(E_2)$ and flat$(J^p(E_2))$ are shown (recall that $E_2 = (\mathtt{a} \mid \mathtt{a})^*$) respectively. If we let $A_2 = \text{nfa}(J^p(E_2))$ and we construct M'_{A_2}, it can be seen that M'_{A_2} has EDA, as Shown in Figure 6.8(c). This suggests that E_2 has an exponential matching time upper bound. It can be confirmed that this upper bound is tight by observing that there are two loops for reading the input string $\mathtt{a}$ from state q_0. As a result of these two loops, while matching the input string $\mathtt{a}^n\mathtt{b}$, every $\mathtt{a}$ symbol can be matched in two ways. This means n $\mathtt{a}$ symbols can be matched in an exponential, in n, number of ways. The $\mathtt{b}$ symbol at the end of the input string will force the matcher to reject every attempt at matching the input string, and consequently force the matcher to try the exponentially many ways of matching the $\mathtt{a}$ symbols.

Next we show why the Java construction provides a more accurate upper bound for the matching time than prioritised Thompson construction. The construction used to convert a regex to pNFA is important, since it determines

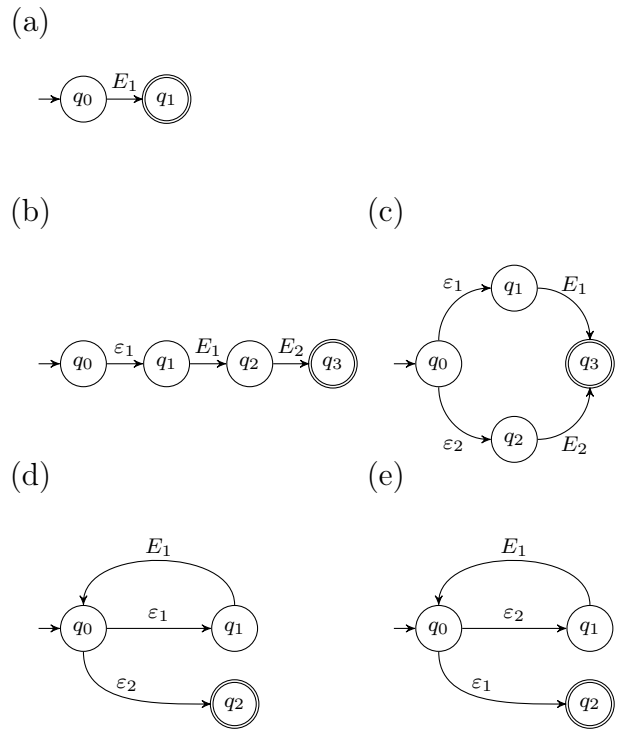

Figure 6.7: The resulting pNFAs of (a) $J^p(E_1)$, (b) $J^p(E_1 \cdot E_2)$, (c) $J^p(E_1 \mid E_2)$, (d) $J^p(E_1^*)$ and (e) $J^p(E_1^{*?})$.

in part on which input strings some regexes exhibit the nonlinear polynomial or exponential matching time, as shown in [23]. Specifically, when constructing a regex, using one construction might result in some input string being capable of triggering the nonlinear polynomial or exponential matching time behaviour, and another construction might result in an entirely different input string being able to do the same.

In more extreme cases, different constructions may even influence whether a regex is vulnerable to nonlinear polynomial or exponential matching time or not. Consider the regex $E_8 = ((\varepsilon \mid \texttt{.}^*)^*(\texttt{a} \mid \texttt{a})^*)$. The pNFAs $Th^p(E_8)$ and $J^p(E_8)$ are given in Figures 6.9 and 6.10, respectively. To understand the difference in vulnerability between these two constructions, compare how each goes about matching the input string $\texttt{a}^n\texttt{b}$. When matching with prioritised Thompson construction, the following path will be taken, $q_0 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_1} q_2 \xrightarrow{\varepsilon_1} q_3 \xrightarrow{\varepsilon_1} q_8 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_2} q_4 \xrightarrow{\varepsilon_1} q_5 \xrightarrow{\cdot} q_6 \xrightarrow{\varepsilon_1} q_5$, and the matcher will continue taking the wildcard loop until the entire input string has been consumed. The matcher will then simply continue to the accept state and accept the input string, backtracking only at states q_5, q_{12} and q_{14}. Since the entire input string was consumed on the wildcard loop, the matching time will be linear in the length of the input string when modeling it using the prioritised Thompson construction. When using the Java construction, the following path is taken,

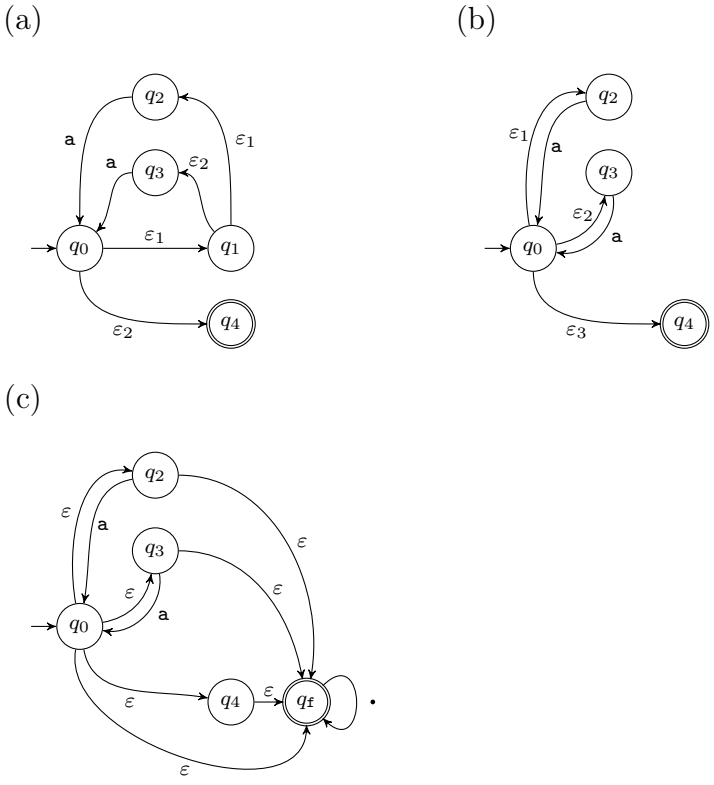

Figure 6.8: The pNFAs (a) $J^p(E_2)$, (b) flat($J^p(E_2)$) where $E_2 = (\mathtt{a} \mid \mathtt{a})^*$ and (c) M'_A where $A = \text{nfa}(J^p(E_2))$.

$q_0 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_1} q_5 \xrightarrow{\varepsilon_1} q_2 \xrightarrow{\varepsilon_1} q_1 \xrightarrow{\varepsilon_1} q_6 \xrightarrow{\varepsilon_1} q_9$, and the matcher will attempt to match $\mathtt{a}^n\mathtt{b}$ with the $(\mathtt{a} \mid \mathtt{a})^*$ subexpression, in exponential time. The difference in vulnerability between the two constructions is due to the fact that in the Java construction, both branches of the alternation in $(\varepsilon \mid .^*)^*$, share an ε-transition, so after the ε-loop has completed, the matcher cannot take the second branch and must continue to match with the next subexpression. The priorities and matching time modeled by the pNFA obtained by the Java construction aligns more accurately with what is observed when measuring the matching time experimentally.

Since the construction used for a regex can affect the vulnerability thereof, we can only accurately state that a regex is vulnerable for a specific construction. Due to this limitation, we will restrict our focus to modeling the matching time of the Java regex matcher.

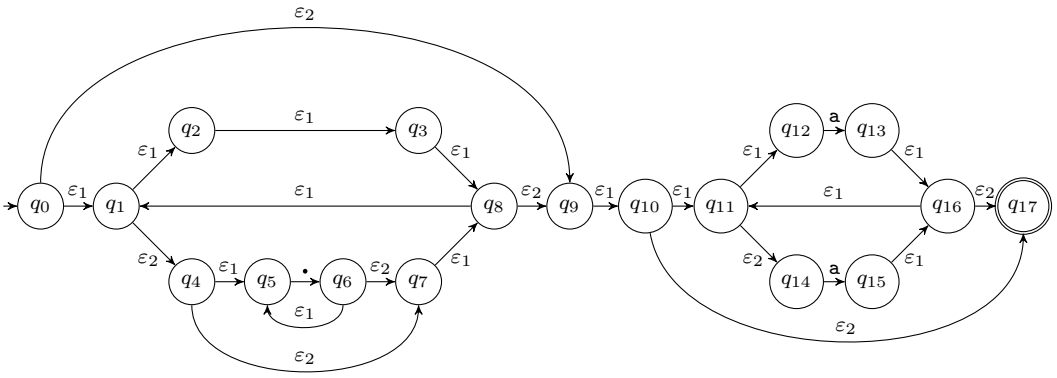

Figure 6.9: The pNFA $Th^p(((\varepsilon \mid \texttt{.}^*)^*(\texttt{a} \mid \texttt{a})^*))$.

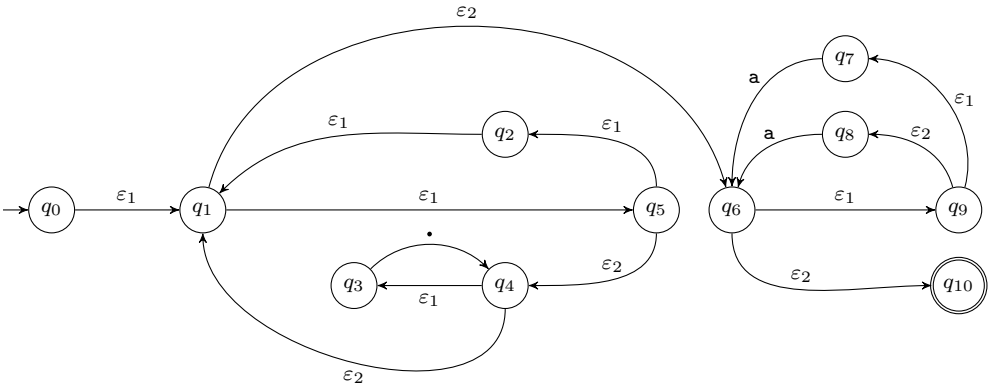

Figure 6.10: The pNFA $J^p(((\varepsilon \mid \texttt{.}^*)^*(\texttt{a} \mid \texttt{a})^*))$.

6.7 Constructing Regexes with Various Asymptotic Matching Times

Now that we have a method of constructing pNFAs capable of modeling the matching behaviour of Java, we construct regexes with various polynomial and exponential growth in matching time.

First, consider the pNFA $A_3 = J^p(E_3)$, where $E_3 = \texttt{a}^*\texttt{a}^*$, as shown in Figure 6.11. From the figure it can be seen that there are two loops reading the symbol a, namely $q_1 \xrightarrow{\varepsilon_1} q_2 \xrightarrow{\texttt{a}} q_1$ and $q_3 \xrightarrow{\varepsilon_1} q_4 \xrightarrow{\texttt{a}} q_3$. Furthermore, the matcher can traverse the pNFA from one loop to the other without reading any input. From the results of Section 2.7.1, specifically Figure 2.3, we have that M'_{A_3} has IDA of degree 2 (and hence we have an upper bound of quadratic

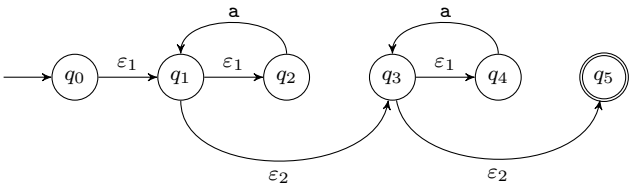

Figure 6.11: The pNFA $J^p(\mathtt{a}^*\mathtt{a}^*)$.

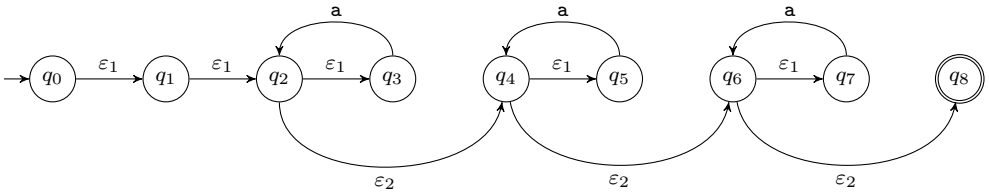

Figure 6.12: The pNFA $J^p(\mathtt{a}^*\mathtt{a}^*\mathtt{a}^*)$.

matching time). Next we consider how the matcher can match the input string $\mathtt{a}^n$. From both the pNFA and regex, it can be seen that the matcher can divide the $\mathtt{a}$ symbols in a linear number of ways between the two $\mathtt{a}^*$ subexpressions. We can therefore conclude that the degree of ambiguity is of order $O(n)$, where n is the length of the input string. Finally, when the matcher is forced to backtrack every path it explores, by giving it a string of the form $\mathtt{a}^n\mathtt{b}$, the matching time is quadratic.

We can increase the degree of the IDA, by adding another $\mathtt{a}^*$ to the regex, as in Figure 6.12. In this case there is a quadratic number of ways in which to divide $\mathtt{a}^n$ between the three $\mathtt{a}^*$ subexpressions in $E_9 = \mathtt{a}^*\mathtt{a}^*\mathtt{a}^*$. Thus the degree of ambiguity is $O(n^2)$ and matching time $O(n^3)$.

It should be clear that it is possible to construct a pNFA for which the matching time is bounded by a polynomial with arbitrary degree, by concatenating the subexpression $\mathtt{a}^*$ multiple times. That is, the regex $\mathtt{a}^* \ldots \mathtt{a}^*$, with d repetitions of $\mathtt{a}^*$, will have IDA of degree $d-1$, that is the degree of ambiguity grows in $O(n^{d-1})$, and hence the matching time in $O(n^d)$.

Extending from the previous idea, we construct a regex for which the pNFA will have exponential worst-case matching time. We need an unbounded number of $\mathtt{a}^*$ in which the matcher can divide the $\mathtt{a}$'s in the input string. This is achieved by the regex $(\mathtt{a}^*)^*$, which has both exponential ambiguity and worst-case matching time.

6.8 Conclusion

This chapter explored the relationship between regexes, NFAs and ambiguity. We also discussed the significance of the regex to pNFA construction used for modeling purposes. We showed regexes with worst-case matching time in $O(n^d)$, for $d \geq 1$, and with exponential worst-case matching time.

Chapter 7

Explaining Regex Vulnerabilities by Example

7.1 Introduction

In Chapter 6, the theory behind vulnerable regexes began to emerge. It was explained at a high level why certain regexes have a bad (i.e., nonlinear) worst-case matching time in Java. In this chapter, the intricacies of regex vulnerabilities are discussed in greater detail, drawing on the concepts explained in Chapters 4 and 6.

In Chapter 6, weak and strong ambiguity of regexes were defined. In this chapter (and whenever the vulnerability of a regex is the main focus) we are less concerned whether a regex E is weakly or strongly ambiguous, and more interested in the maximum rate of growth of the degree of ambiguity. Therefore, for the remainder of this thesis, if we state that a regex is *infinitely ambiguous* (*or exponentially ambiguous*), it can be assumed $\mathrm{sA}_E(n)$ is a nonconstant polynomial (or exponential) function. Moreover, for a word w, if w^n is a word that can be matched in a nonconstant polynomial (or exponential) number of ways by $E(i)$, we say $E(i)$ is infinitely (or exponentially) ambiguous on w^n.

7.2 Vulnerable Regexes

As explained in Chapter 4, when a backtracking matcher fails to match an input string when using higher priority ε-transitions, it backtracks to take the lower priority ε-transitions. When matching an input string with a regex, it is obviously preferred and often assumed that the matching time will be linear in the length of the input strings, however, forcing the matcher to attempt a large number of lower priority transitions, could cause nonlinear matching time. We will consider regexes with nonlinear worst-case matching time and vulnerable regexes to be equivalent. The most straightforward case of regex

vulnerability occurs in regexes with infinite or exponential degree of ambiguity. We discuss this case in Section 7.3. In spite of this, not all regexes with infinite or exponential degree of ambiguity are vulnerable, as shown in Section 7.4.

We will make a single deviation from our theoretical definition of vulnerable regexes in Section 7.5. We do this to investigate regexes with linear matching time in the worst-case, but which should be considered vulnerable in a practical setting.

From Chapter 6, we have that the degree of ambiguity of M'_A, where $A = \text{nfa}(\text{flat}(J^p(E)))$, provides an upper bound for matching time. The fundamental goal of this chapter is to investigate under which conditions this upper bound is tight.

7.3 Vulnerable Regexes with Infinite or Exponential Degree of Ambiguity

When a regex E has infinite or exponential degree of ambiguity, there exists a subexpression $E(i)$ for i as large as possible such that $E(i)$ has the same degree of ambiguity as E. Let w_1^n be a string that can be matched by such an $E(i)$ in a nonconstant polynomial or exponential number of ways. If w_1^n is incorporated as a substring in an input string $w_0 w_1^n w_2$, such that after matching w_0, the matcher is at the initial state corresponding to $E(i)$, and w_2 is selected such that $w_0 w_1^n w_2 \notin \mathcal{L}(E)$, then the matcher is forced the to try each of the many ways of matching w_1^n with $E(i)$, causing a nonlinear matching time.

Next we provide two examples illustrating these ideas.

The pNFA $J^p(E)$, when $E = \mathtt{a}^*\mathtt{a}^*$, has infinite ambiguity of degree 1, as shown in Chapter 6. Also, $\text{flat}(J^p(E))$ is shown in Figure 7.1. Using $\text{flat}(J^p(E))$, we construct the backtracking trees obtained during matching. The number of states in these backtracking trees models the duration of the matching time. Also, the growth in size of these backtracking trees, in relation to the length of the input string, models the rate of growth of the matching time. The backtracking trees obtained when matching `a`, `ab`, `aa` and `aab` are shown in Figures 7.2 and 7.3 respectively. The increase in number of states visited when matching `ab` or `aab`, is caused by E being infinitely ambiguous on $\mathtt{a}^n$, and the suffix `b` forcing the matcher to try numerous ways of matching the prefixes. The number of states in backtracking trees, and therefore the matching time, on input $\mathtt{a}^n\mathtt{b}$, and the worst-case matching in general, grows quadratically.

The regex $E = (\mathtt{a} \mid \mathtt{a})^*$ has exponential degree of ambiguity, as shown in Chapter 6. The pNFAs $J^p(E)$ and $\text{flat}(J^p(E))$, are given in Figure 6.8, and the backtracking trees for matching `a`, `ab` and `aab` in Figures 7.4 and 7.5. Matching time when using $\mathtt{a}^n\mathtt{b}$ as input, and in the worst-case in general, grows exponentially.

Unfortunately, infinite or exponential degree of ambiguity in regexes is not

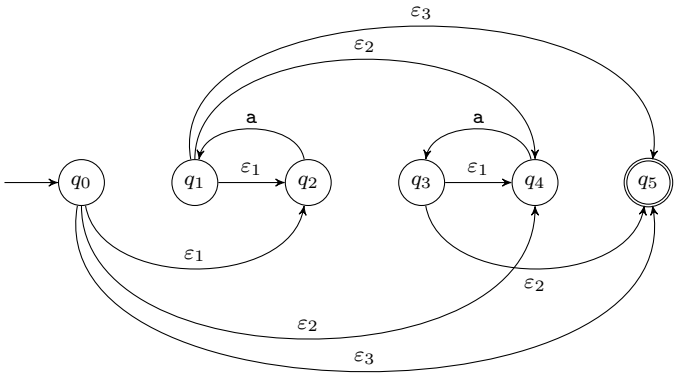

Figure 7.1: The pNFA flat($J^p(\texttt{a}^*\texttt{a}^*)$).

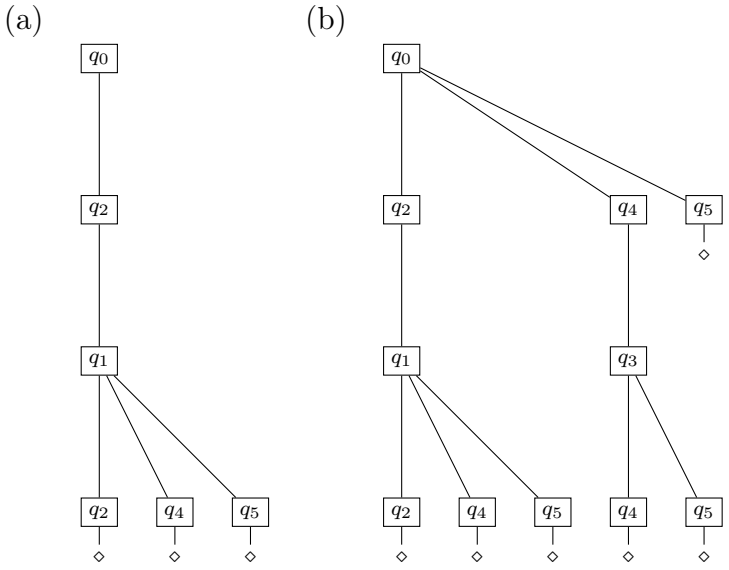

Figure 7.2: The backtracking trees produced when matching (a) `a` and (b) `ab` with the pNFA flat($J^p(\texttt{a}^*\texttt{a}^*)$).

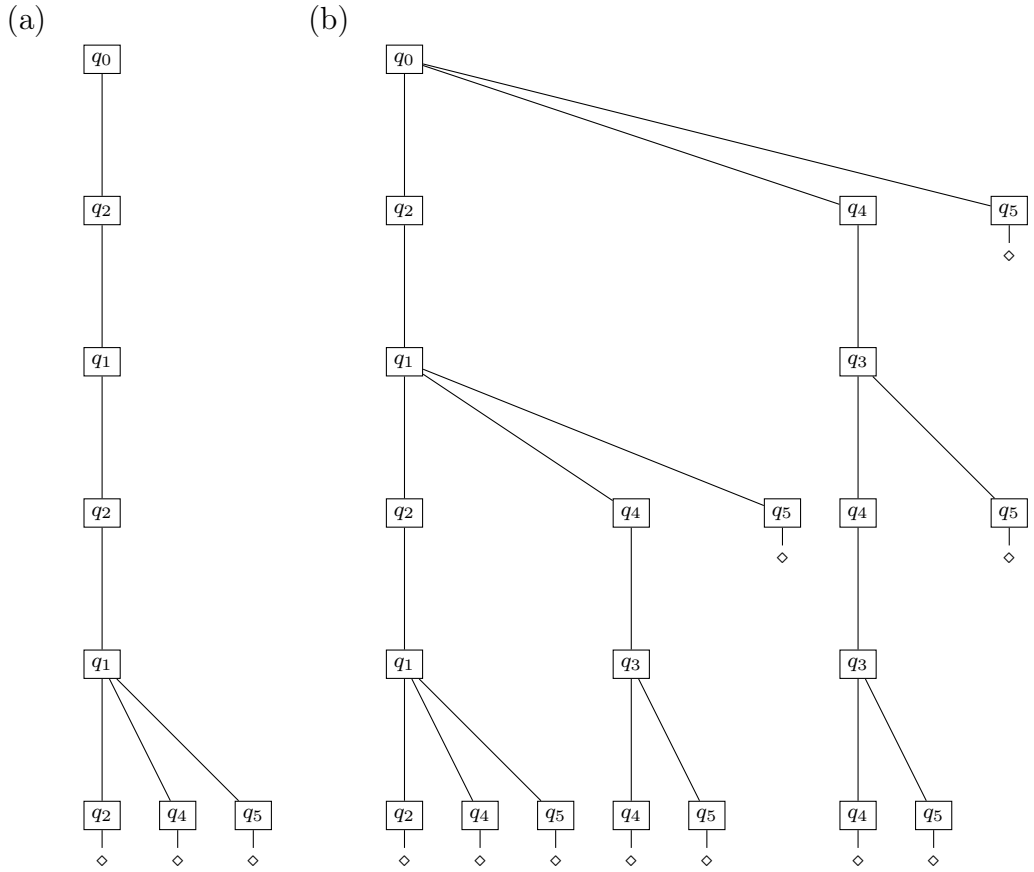

Figure 7.3: The backtracking trees produced when matching (a) `aa` and (b) `aab` with the pNFA flat(J^p(`a*a*`)).

sufficient for regex vulnerability. This was alluded to in Chapter 6, when it was shown that some regexes with exponential degree of ambiguity can match all strings in linear time.

7.4 Harmless Regexes with Infinite or Exponential Degree of Ambiguity

As mentioned before, a regex with infinite or exponential degree of ambiguity might not be vulnerable, due to how matching with some subexpressions is prioritised over matching with others. Consider for example a regex $E = (E_1 \,|\, E_2 \,|\, \ldots \,|\, E_n \,|\, E_{\mathtt{v}})$, where $E_{\mathtt{v}}$ being vulnerable, while each E_i has linear matching time, and assume $\mathcal{L}(E_1 \,|\, \ldots \,|\, E_n) = \Sigma^*$. Because of the prioritisation of matching with subexpressions, E will match every input string in linear time, and is consequently not vulnerable. A specific example of this nature was introduced in Chapter 6, where $E = (\mathtt{.*}\,|(\mathtt{a^*})^*)$. The pNFAs $J^p(E)$ and flat($J^p(E)$),

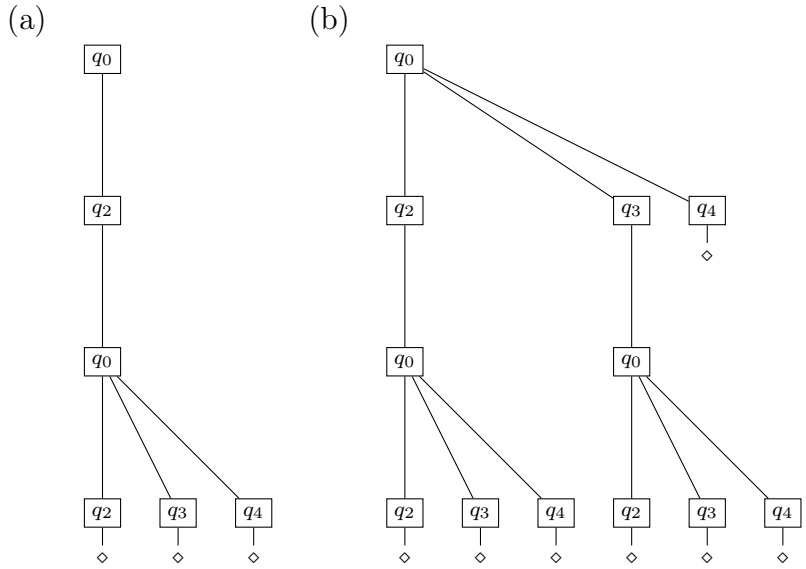

Figure 7.4: The backtracking trees produced when matching (a) `a` and (b) `ab` with the pNFA flat($J^p((\mathtt{a}\,|\,\mathtt{a})^*)$).

shown in Figure 7.7, model the behaviour of the Java regex matcher. Inspecting these pNFAs shows that a higher priority is given to matching with the subexpression $E(2) = \mathtt{.}^*$ than with $E(4) = (\mathtt{a}^*)^*$. Consequently, since $E(2)$ accepts all strings in linear time, the matching time will be linear for any input string. Also, the backtracking trees produced when matching the input strings `ab`, `aab` and $\mathtt{a}^n\mathtt{b}$ with flat($J^p(E)$), are shown in Figure 7.6. The number of states in the backtracking trees grows linearly with respect to the length of the input strings, which confirms linear matching time.

The harmlessness of certain regexes with infinite or exponential degree of ambiguity does not necessarily rely on the highly ambiguous subexpressions being unreachable. A regex $E = E_0E_1$, in which E_0 has infinite or exponential degree of ambiguity, is not necessarily vulnerable, for example in the regex $E = (\mathtt{a}\,|\,\mathtt{a})^*\mathtt{.}^*$, the subexpression $(\mathtt{a}\,|\,\mathtt{a})^*$ with exponential degree of ambiguity is reachable, but it is impossible to force the matcher to backtrack to try the exponentially many ways of matching an input string $\mathtt{a}^n$. For a more involved example, consider the regex $E = \mathtt{a\{}l\mathtt{\}.}^*\,|(\mathtt{a}\,|\,\mathtt{a})^*$, for a constant integer l. The subexpression with exponential degree of ambiguity, $(\mathtt{a}\,|\,\mathtt{a})^*$, is indeed reachable, however, only on input strings of the form $\mathtt{a}^nw$, where $n < l$ and w is selected such that $\mathtt{a}^nw \notin \mathcal{L}(\mathtt{a\{}l\mathtt{\}.}^*)$. Consequently, for l small enough, this regex is not vulnerable, since the matching time cannot grow large enough (in spite of its exponential growth) before the subexpression with exponential degree of ambiguity becomes unreachable and matching completes in linear time.

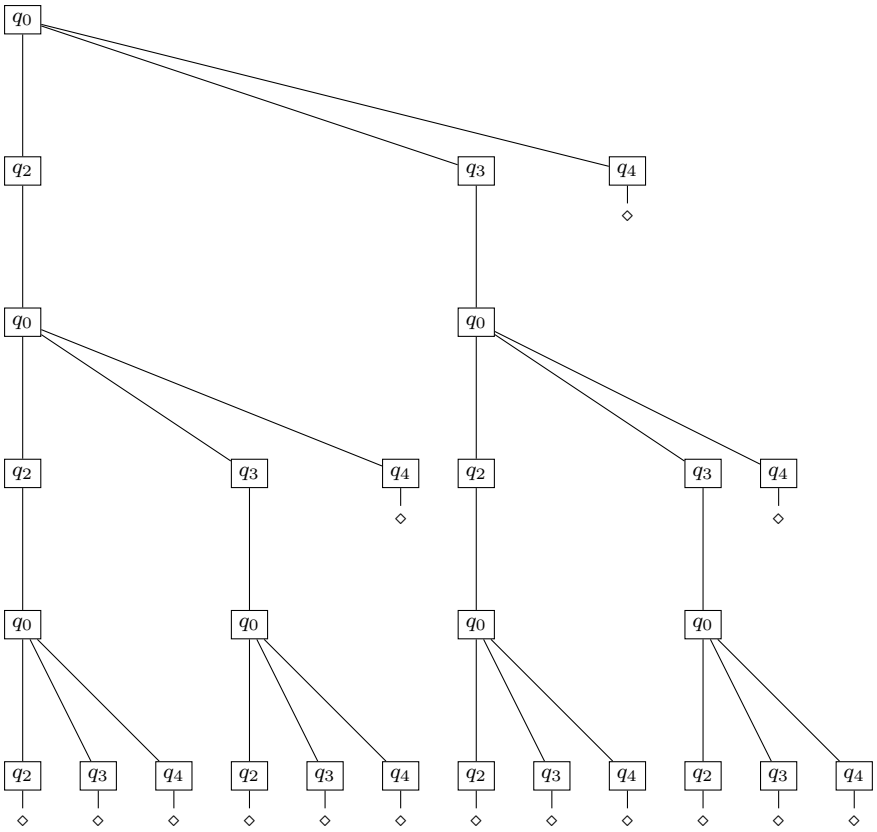

Figure 7.5: The backtracking tree produced when matching `aab` with the pNFA $\text{flat}(J^p((\texttt{a} \mid \texttt{a})^*))$.

7.5 Vulnerable Regexes with Constant Degree of Ambiguity

It might be tempting to declare all regexes with constant degree of ambiguity as harmless, but this may not necessarily be the case. As a counterexample, consider the regex $E = E_1E_2$, where the subexpression E_1 has exponential degree of ambiguity, with the string w^n being exponentially ambiguous and $\mathcal{L}(E_2) = \emptyset$. The regex E is not ambiguous, indeed $\mathcal{L}(E) = \emptyset$. In spite of this, the regex is vulnerable. If we assume the regex matcher does not simplify E to $\emptyset$, matching the regex with the input string w^n will exhibit exponential matching time. This occurs due to the matcher still attempting the exponentially many ways of matching w^n with E_1, while rejecting each way due to the subexpression E_2.

There also exist regexes with constant degree of ambiguity, but for which this constant is so large, that the matching time can become extremely long, albeit linear in the length of the input string. In contrast to our theoretical definition of vulnerable regexes, of Section 7.2, it is reasonable to consider

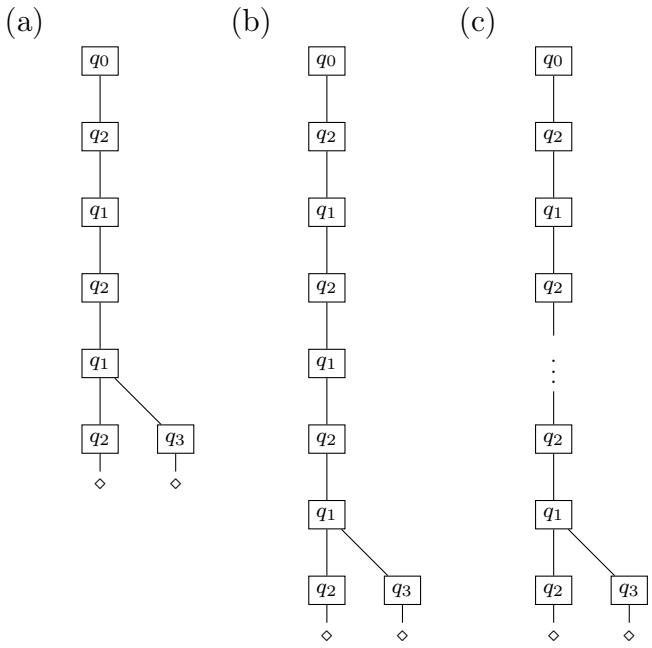

Figure 7.6: The backtracking trees produced when matching (a) `ab`, (b) `aab` and (c) $\texttt{a}^n\texttt{b}$ with the pNFA flat($J^p(E)$), where $E = \texttt{.}^*\,|(\texttt{a}\,|\,\texttt{a})^*$.

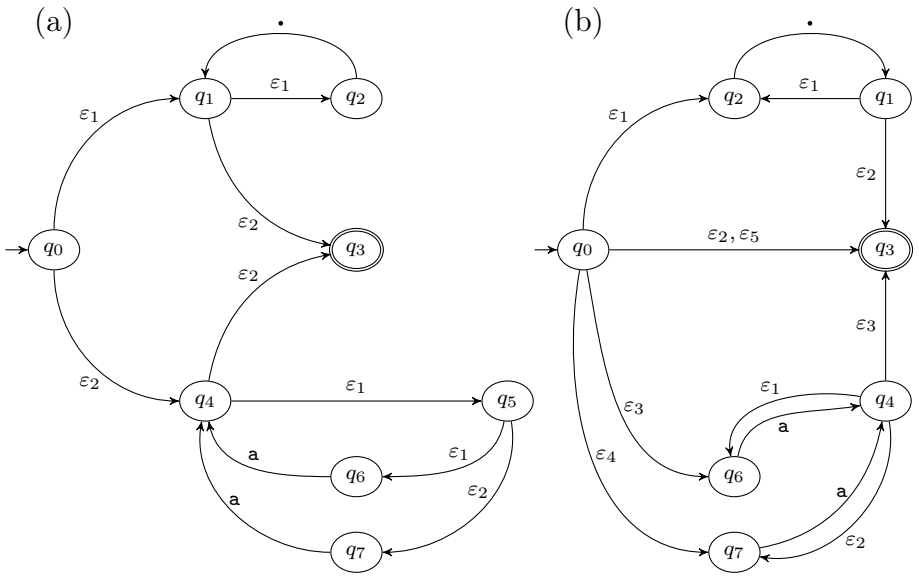

Figure 7.7: The pNFAs (a) $J^p(E)$ and (b) flat($J^p(E)$) for $E = \texttt{.}^*\,|(\texttt{a}\,|\,\texttt{a})^*$.

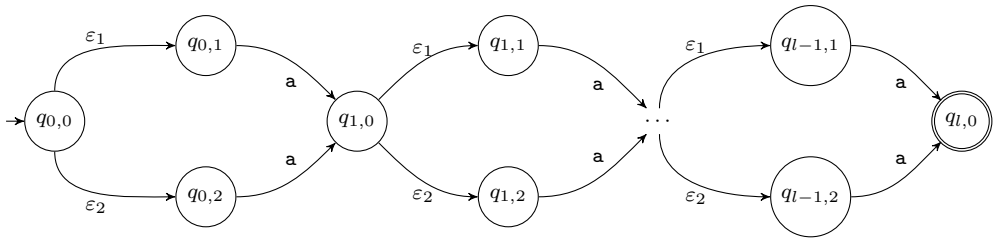

Figure 7.8: The pNFA flat(J^p((a | a){l})).

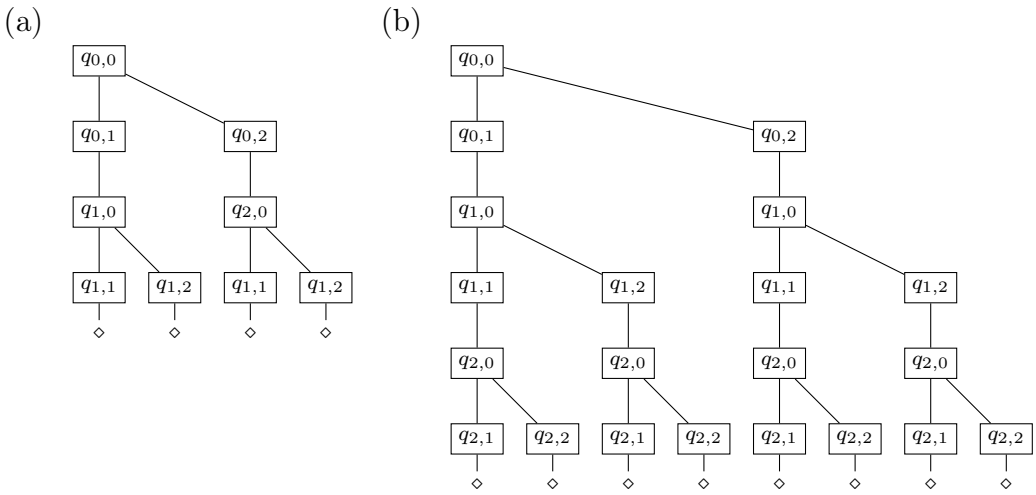

Figure 7.9: The backtracking trees produced when matching (a) ab and (b) aab with the pNFA flat(J^p((a | a){l})).

these regexes as vulnerable, at least in a practical sense. Consider for example a regex such as $E = (\mathtt{a} \mid \mathtt{a})\{l\}$. This regex has constant degree of ambiguity, which can be verified quickly, by noting the regex does not match input strings longer than l. In spite of this, when matching input strings of the form $\mathtt{a}^n\mathtt{b}$, for $n < l$, the matching time will grow exponentially in n up to $n = l$. If l is sufficiently large, the matching time will become large enough for the regex to be considered vulnerable. The pNFA flat($J^p(E)$), is given in Figure 7.8, and backtracking trees when matching ab and aab in Figure 7.9. For this exponential growth to occur, the b suffix is unnecessary, and exponential matching time also occurs for input string $\mathtt{a}^n$, where $n < l$.

7.6 Conclusion

In this chapter we explored, via examples, the relationship between infinite and exponential degree of ambiguity of regexes and the property of a regex being vulnerable.

Chapter 8

Deciding Worst-case Matching Time

8.1 Introduction

In this chapter, we investigate how the results from Chapters 2, 4 and 6 can be used to determine the worst-case matching time of regexes. This will ultimately allow us to decide if a regex is vulnerable, or not. We provide two analyses, a polynomial algorithm allowing us to calculate an upper bound for the worst-case matching time of a regex, and an exponential algorithm for determining a tight bound.

8.2 Matching Time Analysis

In this section we will describe two different methods for analysing the matching time of a regex. As stated in Chapter 6, we focus mainly on the matching time behaviour of the Java regex matcher. Nonetheless, the results returned by simple analysis are relevant for other backtracking matchers as well, as explained in Section 8.2.1. Since we are mainly interested in the Java regex matcher, the analyses will operate over pNFAs constructed with the Java construction, introduced in Chapter 6. Therefore, when analysing regex E, our analysis starts by constructing the pNFA $J^p(E)$.

As explained in Section 4.4.1, when modeling backtracking matching, we need to prevent the reuse of ε-transitions between reading input symbols. In our analyses this is achieved by converting the pNFA $J^p(E)$ to the flattened pNFA $\text{flat}(J^p(E))$.

8.2.1 Simple Analysis

Simple analysis provides an upper bound for the matching time of a regex by calculating the degree of ambiguity of an NFA obtained from the regex under

consideration. Simple analysis relies on the result obtained in Chapter 6, showing that the degree of ambiguity of an NFA (obtained from a regex E), can be related to worst-case matching time with the regex E. Since simple analysis only calculates the degree of ambiguity, the upper bound returned thereby is valid, although not necessarily tight, for any backtracking matcher that uses a regex to automaton construction that preserves the degree of regex ambiguity.

To analyse the regex E, the NFA $A = \text{nfa}(\text{flat}(J^p(E)))$ is constructed. From A, M'_A is constructed (as defined in Chapter 6), since the degree of ambiguity of M'_A serves as an upper bound for the matching time of E. The degree of ambiguity of M'_A can be calculated by, for example, using the algorithms provided in [13; 14], which will allow simple analysis to be performed in polynomial time, as shown in Theorem 8.1. Recall that $|A|_\delta$ denotes the number of transitions in A.

Theorem 8.1 ([14])**.** Let A be an ε-loop free NFA, for which all states are reachable. Then

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

What follows is a high-level description of algorithms that can be used to detect IDA and EDA in an NFA. For a more in-depth discussion, consult [14]. For detecting IDA in an NFA, the product $A^3 = A \times F \times A \times F \times A$ is calculated, where F is a transducer used to eliminate some ε-transitions that cause false positive cases of IDA and EDA. In the NFA A^3, the states form 3-tuples (p_0, p_1, p_2), where $p_i \in Q$, and the transition function is a function $\delta : Q^3 \times \Sigma^\varepsilon \times Q^3 \to \mathbb{N}$, defined as $\delta((p_0, p_1, p_2), \alpha, (p'_0, p'_1, p'_2)) = \prod_{i=0}^{2}(\delta(p_i, \alpha, p'_i))$ if $\delta(p_i, \alpha, p'_i)$ is defined for each $i \in \{0, 1, 2\}$, and otherwise $\delta((p_0, p_1, p_2), \alpha, (p'_0, p'_1, p'_2))$ is undefined. The NFA A^3 is then explored to search for a path from a state (p, p, q) to (p, q, q) where $p \neq q$. Such a path in A^3 indicates the existence of a path from state p to p, as well as a path from state p to q and a path from q to q, all on the same input string. From the results in Chapter 2, we know this is an indication of IDA of degree (at least) 1. The highest degree of IDA in A can be determined by finding the longest path, linking pairs of states (p_i, p_i, q_i) and (p_i, q_i, q_i) in A^3, by using nonempty strings. For example, a path $(p_0, p_0, q_0) \xrightarrow{w_0} (p_0, q_0, q_0) \xrightarrow{w_1} (p_1, p_1, q_1) \xrightarrow{w_2} (p_1, q_1, q_1)$ indicates IDA of degree (at least) 2, if $w_0, w_2 \neq \varepsilon$. As an example, consider the regex $E = \texttt{a}^*\texttt{a}^*\texttt{bc}^*\texttt{c}^*$. The NFA $A = \text{nfa}(\text{flat}(J^p(E)))$ is shown in Figure 8.1(a). Comparing A to Figure 2.5 should make it clear that M'_A has IDA of degree 3. We provide some of the states and transitions of the NFA $(M'_A)^3$ in Figure 8.1(b). From the figure we see that there exists a path from

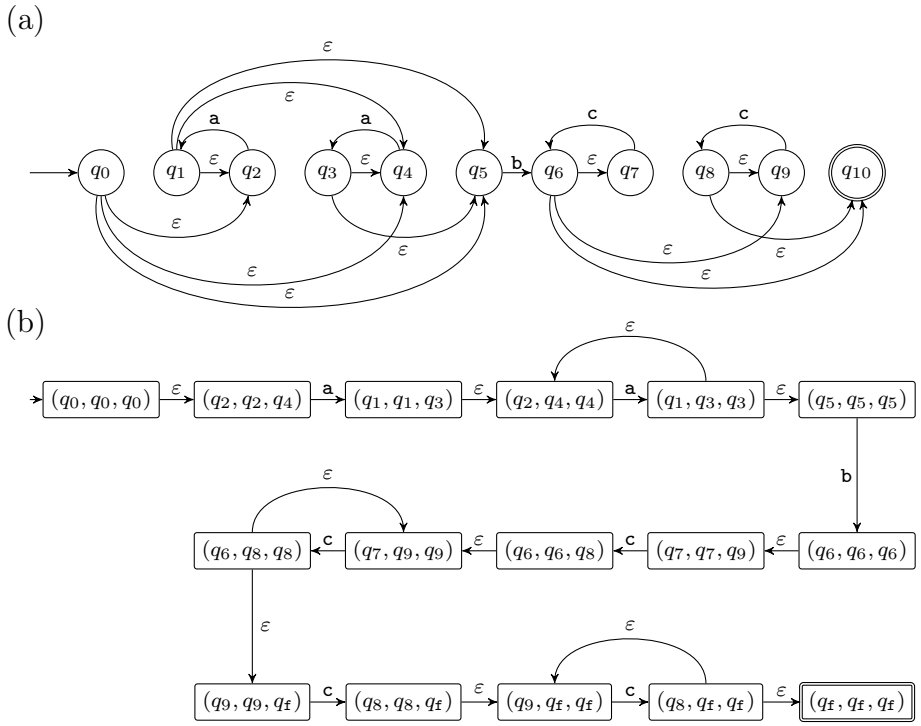

Figure 8.1: (a) The NFA $A = \text{nfa}(\text{flat}(J^p(E)))$ with $E = \texttt{a}^*\texttt{a}^*\texttt{bc}^*\texttt{c}^*$ and (b) some of the states and transitions of $(M'_A)^3$.

state (q_2, q_2, q_4) to (q_2, q_4, q_4) on a, from (q_7, q_7, q_9) to (q_7, q_9, q_9) on c, and from (q_9, q_9, q_f) to (q_9, q_f, q_f) on c. Furthermore, these paths can all be incorporated into a single larger path. Therefore, M'_A has IDA of degree 3. Simple analysis thus provides an upper bound of cubic matching time for E, which is tight in this case.

Similarly, for detecting exponential degree of ambiguity in an NFA, $A^2 = A \times F \times A$ is calculated. In the NFA A^2, the states are tuples (p_0, p_1) where $p_i \in Q$ and the transition function is a function $\delta : Q^2 \times \Sigma^\varepsilon \times Q^2 \to \mathbb{N}$, defined as $\delta((p_0, p_1), \alpha, (p'_0, p'_1)) = \prod_{i=0}^{1}(\delta(p_i, \alpha, p'_i))$ if $\delta(p_i, \alpha, p'_i)$ is defined for each i, otherwise $\delta((p_0, p_1), \alpha, (p'_0, p'_1))$ is undefined. Within each strongly connected component (a subset of states that are all mutually reachable) of A^2 we search for a path containing at least one symbol transition between states of the form (p, p) and (q, q'), such that $q \neq q'$. The presence of this path indicates two distinct loops from p to itself in A, while reading the same nonempty input string. From the results in Chapter 2, we know this is an indication of EDA. To illustrate the detection of EDA in an NFA, we give $A = \text{nfa}(\text{flat}(J^p(E)))$, for $E = \texttt{.}^* \,|(\texttt{a} \,|\, \texttt{a})^*$ in Figure 8.2. Noting the two loops from state q_4 to itself while reading the input symbol a should be enough to convince us that A has EDA, but we also provide some of the states and transitions of the NFA $(M'_A)^2$ in Figure 8.2(b). It can be seen that in $(M'_A)^2$ there is a strongly connected

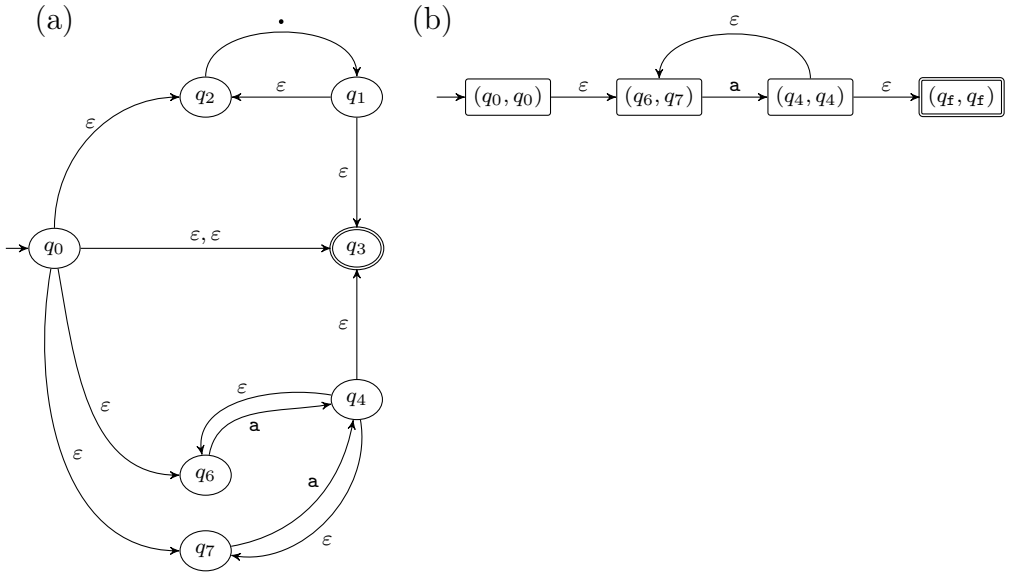

Figure 8.2: (a) The NFA $A = \text{nfa}(\text{flat}(J^p(E)))$ with $E = \texttt{.}^* \,|\, (\texttt{a} \,|\, \texttt{a})^*$ and (b) some of the states and transitions of $(M'_A)^2$.

component containing the states (q_6, q_7) and (q_4, q_4). In this strongly connected component, the state (q_4, q_4) can be reached from (q_6, q_7) by reading the input string `a`. This indicates two loops in M'_A, from state q_4 to itself, while reading `a` and therefore our algorithm for detecting exponential degree of ambiguity will determine M'_A indeed has EDA. Simple analysis will therefore provide an upper bound of exponential matching time for E. In contrast to this, in Chapter 7 we have seen that E has linear matching time.

8.2.2 Full Analysis

When priorities of ε-transitions are ignored, information is lost, and consequently, converting the pNFA $\text{flat}(J^p(E))$ to the NFA $\text{nfa}(\text{flat}(J^p(E)))$, causes the imprecise results of simple analysis. As illustrated in Chapter 4, a backtracking matcher prioritises certain subexpressions over others and it was shown in Chapter 7 that these priorities may alter matching time, by rendering some subexpressions unreachable. This usually happens due to the matcher always finding a match with some higher priority subexpression. In this section we describe an analysis technique that yields precise results, by removing these unreachable states. We refer to this analysis as the *full analysis*.

In order to obtain a precise result from the analysis, we need to incorporate the priorities of the ε-transitions therein. To do this, we define the concept of an *unprioritised pNFA (upNFA)*. From a pNFA A we obtain the upNFA, denoted by $\text{up}(A)$, by keeping track in each state of $\text{up}(A)$ the states of A the matcher could have been in by following a path with higher priority for the

same input string. This is done by performing a type of subset construction of all the states reachable via a higher priority path. To simplify the construction of up(A), we assume A has no ε-loops. If this is not the case, we convert A to flat(A) and construct up(flat(A)) instead. The formal definition of a upNFA is given in Definition 8.1. An example of converting $J^p(E)$ to flat($J^p(E)$) and then to up(flat($J^p(E)$)), is given in Figure 8.3, for $E = \texttt{.}^* \,|\, (\texttt{a} \,|\, \texttt{a})^*$.

For the pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, the states in a upNFA up(A) are tuples (p, P), where $p \in Q_1 \cup Q_2$ and $P \subseteq Q_1$. We then remove from up(A) the states with a P such that every $w \in \Sigma^*$ can be accepted by traversing a path in A starting at some $p' \in P$. Let us refer to up(A) with these unreachable states removed, as A'. After the unreachable states have been removed from up(A), the matching time is determined, by constructing $M'_{A'}$ and using it to determine an upper bound for the matching time, in the same way as for simple analysis.

In order to remove the unreachable states we are effectively required to decide if an automaton accepts the entire language Σ^*. To decide this is to solve the NFA-universality problem, which is PSPACE-complete, as shown in [33]. Consequently, full analysis is much more computationally expensive than simple analysis.

To understand the difference the upNFA makes, compare the upNFA $A_1 = \text{up}(J^p(E_1))$ in Figure 8.3, for the harmless regex $E_1 = \texttt{.}^* \,|\, (\texttt{a}^*)^*$, to the upNFA $A_2 = \text{up}(J^p(E_2))$ in Figure 8.4, for the vulnerable regex $E_2 = (\texttt{a}^*)^* \,|\, \texttt{.}^*$. Assume all the dashed states, which indicate the unreachable states, have already been removed from both A_1 and A_2. The result is, in A_1, almost all that remains is a wildcard loop and therefore M'_{A_1} will have infinite ambiguity of degree 1. With this information, a tight upper bound of linear matching time can be assigned to E_1. Conversely, in M'_{A_2} there will still be multiple paths from state $(q_5, \{q_3, q_6\})$ to itself while reading the input string $\texttt{a}$, indicating EDA and since these states are indeed reachable, the regex is assigned a tight upper bound of exponential matching time and is therefore vulnerable.

Definition 8.1. Let $A := (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, then up($A$) is the NFA given by $(Q', \Sigma, q'_0, \delta', F')$, where:

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

(ii) $q'_0 = \{(q_0, \emptyset)\}$ and $F' = (F \times \mathcal{P}(Q_1)) \cap Q'$;

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

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

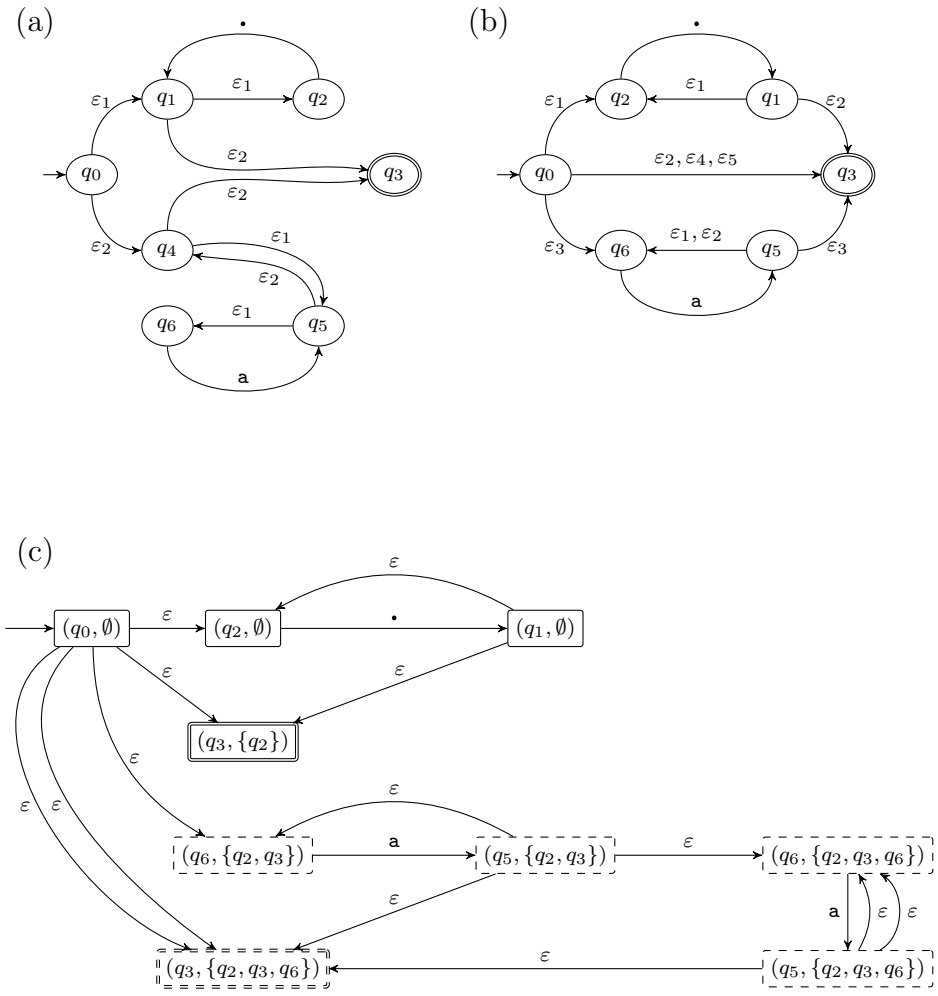

Figure 8.3: The pNFAs (a) $J^p(E)$, (b) flat($J^p(E)$) and (c) the unprioritised pNFA for flat($J^p(E)$) for $E = \texttt{.}^*|(\texttt{a}^*)^*$. Dashed states indicate the unreachable states (states Q'' in Definition 8.1) that should be removed.

8.3 Conclusion

In this chapter we brought together results from previous chapters in order to develop static analysis techniques for regexes. We described simple and full analysis, i.e., two ways of estimating worst-case matching time for a given regex. These methods provide an upper and precise bound respectively. By using our static analysis techniques, developers can determine whether the regexes used in their software is safe to use or not.

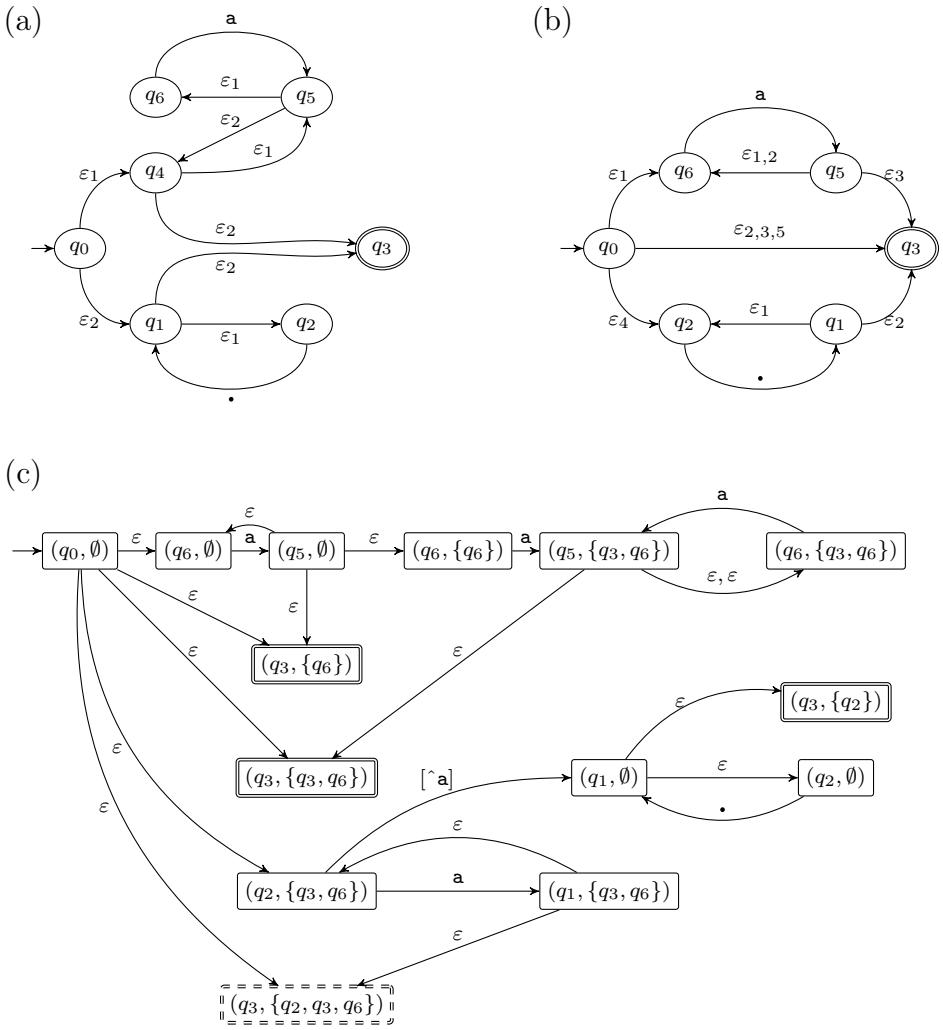

Figure 8.4: The pNFAs (a) $J^p(E)$, (b) flat$(J^p(E))$ and (c) the unprioritised pNFA for flat$(J^p(E))$ for $E = (\mathtt{a}^*)^* \mid \mathtt{.}^*$ we show. Dashed states indicate the unreachable states (states Q'' in Definition 8.1) that should be removed. Recall the notation [^a] means *any symbol but* a

Chapter 9

Triggering Worst-Case Matching Time

9.1 Introduction

Along with knowing whether or not a regex is vulnerable, it is also valuable to know for which input strings the nonlinear polynomial, or exponential matching time is triggered. Such strings can ultimately be used to prove the vulnerability of a regex.

9.2 Exploit Strings

We refer to strings that trigger the worst-case matching time of a vulnerable regex as *exploit strings*. Exploit strings usually achieve their goal by forcing the matcher to try the numerous ways of matching with some ambiguous subexpression $E(i)$ of a regex E.

9.3 Building Exploit Strings

To explain how exploit strings are constructed for vulnerable regexes, we use the example regex $E_0E_1E_2$, which we assume is vulnerable. Let $s_0 \in \mathcal{L}(E_0)$, $r \notin \mathcal{L}(E_1E_2)$, and assume E_1 is exponentially ambiguous on w_1^n for $n > 0$.

Exploit strings are made up of four types of parts, namely the *prefix*, the *pumps*, the *pump separators* and the *suffix*. An exploit string is typically of the form $s_0w_0^{n_0}s_1w_1^{n_1}s_2 \dots w_k^{n_k}r$, where $n_0, n_1, \dots, n_k \in \mathbb{N}$, where s_0 is the prefix, w_i and s_j where $i, j \in \mathbb{N}$, $j > 0$, are the pumps and pump separators, respectively, and lastly, r is the suffix. These parts are explained in the following sections.

9.3.1 Prefix

The prefix of an exploit string is the substring that allows the matcher to reach an infinitely, or exponentially ambiguous subexpression. In our example regex $E_0E_1E_2$, we need to force the matcher to reach the ambiguous subexpression E_1. This can be achieved by using $s_0 \in \mathcal{L}(E_0)$ as a prefix of the exploit string.

9.3.2 Pumps

The main goal of a pump is to create a large number of backtracking possibilities for a matcher. For our example vulnerable regex we use the string w_1 as a pump, since it is assumed that E_1 is exponentially ambiguous on w_1^n.

Note that in some cases it is required to have multiple pumps. For example, consider a regex of the form ${E_0}^*{E_1}^*{R_0}^*{R_1}^*$, where $w_0 \in \mathcal{L}(E_0) \cap \mathcal{L}(E_1)$ and $w_1 \in \mathcal{L}(R_0) \cap \mathcal{L}(R_1)$ and $\mathcal{L}(E_0) \cap \mathcal{L}(E_1) \cap \mathcal{L}(R_0) \cap \mathcal{L}(R_1) = \emptyset$, i.e., $w_0 \neq w_1$. This regex has infinite ambiguity of degree at least 2 and the worst-case matching time is (at least) cubic in the length of the input string, if it is, for example, the case that we can find a string r, such that all strings having r as suffix are not in $\mathcal{L}({E_0}^*{E_1}^*{R_0}^*{R_1}^*)$. Thus in this case strings of the form $w_0^{n_0}w_1^{n_1}r$ can be used as exploit strings (and thus we can use the empty string as prefix and pump separator).

9.3.3 Pump Separators

The regex ${E_0}^*{E_1}^*{R_0}^*{R_1}^*$ was used above to motivate the need (in some cases) for more than one pump in an exploit string, but the subexpressions ${E_0}^*{E_1}^*$ and ${R_0}^*{R_1}^*$ do not have to appear in succession for the regex to have infinite ambiguity of degree (at least) 2. For example, the regex ${E_0}^*{E_1}^*R{F_0}^*{F_1}^*$, where $\mathcal{L}(E_0) \cap \mathcal{L}(E_1) \neq \emptyset$, $\mathcal{L}(F_0) \cap \mathcal{L}(F_1) \neq \emptyset$ and $\mathcal{L}(R) \neq \emptyset$ will also have infinite ambiguity of degree (at least) 2. For this regex, the pumps can be selected from $\mathcal{L}(E_0) \cap \mathcal{L}(E_1)$ and $\mathcal{L}(F_0) \cap \mathcal{L}(F_1)$ respectively, and any string in $\mathcal{L}(R)$ can be used as pump separator.

9.3.4 Suffixes

Exploit strings are constructed with a suffix which forces the matcher to backtrack and revisit all the backtracking possibilities created by the pumps. This is usually achieved by appending some string that is rejected by subexpressions in which the final pump was matched and also by all subsequent subexpressions. For example, with $E_0E_1E_2$, we can use $r \notin \mathcal{L}(E_1E_2)$ as suffix of the exploit string.

Note that it is not required that exploit strings are rejected by the matcher. Consider for example a regex of the form $E_0E_1 \mid E_2$, where E_0 is a subexpression with exponential degree of ambiguity. Assume E_0 is exponentially ambiguous

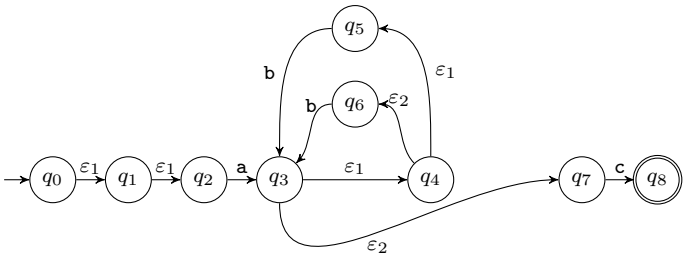

Figure 9.1: The pNFA $J^p(E)$ where $E = \mathtt{a}(\mathtt{b} \mid \mathtt{b})^*\mathtt{c}$.

on w_0^n, $w_0^n r \notin \mathcal{L}(E_0E_1)$ and $w_0^n r \in E_2$. Then exploit strings of the form $w_0^n r$ will trigger exponential matching time, since matching with E_0E_1 is given a higher priority than with E_2. A specific example of this type is given by $E = (\mathtt{a} \mid \mathtt{a})^* \mid \mathtt{.}^*$, for which strings of the form $\mathtt{a}^n\mathtt{b}$ can be used as exploit strings.

Example 9.1. In this example we show how to build exploit strings for the vulnerable regex $E = \mathtt{a}(\mathtt{b} \mid \mathtt{b})^*\mathtt{c}$. The pNFA $J^p(E)$ is given in Figure 9.1. From Chapter 6 it should be clear that the subexpression $(\mathtt{b} \mid \mathtt{b})^*$ has exponential degree of ambiguity. In order to allow the matcher to reach $(\mathtt{b} \mid \mathtt{b})^*$, we use $\mathtt{a}$ as prefix in our exploit strings. The suffix needs to force the matcher to try all the possible ways of matching the exploit strings, and thus the string $\mathtt{d}$ will suffice. We thus use $\mathtt{ab}^n\mathtt{d}$ as exploit strings.

Example 9.2. When considering the slightly more complicated vulnerable regex $E = \mathtt{ab}^*\mathtt{b}^*\mathtt{cd}^*\mathtt{d}^*\mathtt{e}$, the previous discussions in this chapter should make it clear that $\mathtt{ab}^{n_0}\mathtt{cd}^{n_1}\mathtt{f}$ can for example be used as exploit strings.

9.4 Attack Automata

The problem of constructing exploit strings for vulnerable regexes has been investigated more thoroughly in [34]. In this paper it is described how an automaton can be constructed to recognise the language of all exploit strings for a specific vulnerable regex. These automata are referred to as *attack automata*.

Using an approach similar to the one described in [34], we can construct attack automata for vulnerable regexes. For $E_1(E_2 \mid E_3)^*E_4$ for example, we can construct an attack automaton as follows. Automata A_p, A_c and A_s are created such that $\mathcal{L}(A_p) = \mathcal{L}(E_1)$, $\mathcal{L}(A_c) = \mathcal{L}(E_2) \cap \mathcal{L}(E_3)$ and $\mathcal{L}(A_s) = \overline{\mathcal{L}((E_2 \mid E_3)^*E_4)}$, where $\overline{\mathcal{L}(E)}$ is used to denote the complement of the language $\mathcal{L}(E)$. The resulting attack automata A_a is formed by using A_p, A_c and A_s, such that $\mathcal{L}(A_a) = \mathcal{L}(A_p) \cdot (\mathcal{L}(A_c))^+ \cdot \mathcal{L}(A_s)$.

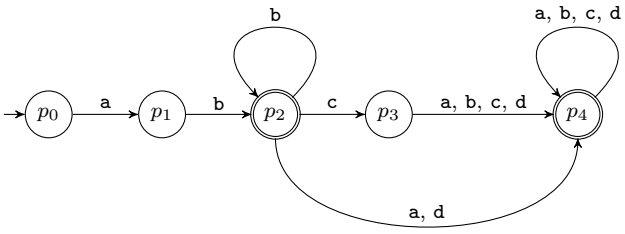

Figure 9.2: An attack automaton for the vulnerable regex $E = \mathtt{a}(\mathtt{b} \mid \mathtt{b})^*\mathtt{c}$ for which $J^p(E)$ is shown in Figure 9.1.

An attack automaton for the regex used in Example 9.1, $\mathtt{a}(\mathtt{b} \mid \mathtt{b})^*\mathtt{c}$, is given in Figure 9.2. Note, this attack automaton is constructed over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, but it is trivial to extend it to an arbitrary alphabet.

9.5 Conclusion

In this chapter we discussed the process of constructing strings capable of triggering the worst-case matching time of vulnerable regexes. Examples were given to show how these exploit strings can be used to confirm the vulnerability of a given regex.

Chapter 10

Experimental results

10.1 Introduction

We can test the effectiveness of the analyses developed in Chapter 8 by analysing regexes used in practice. If an analysis labels a regex as vulnerable, it should be possible to trigger exponential matching time in the Java matcher while matching the regex with some input string. In this chapter, we present the results obtained when experimentally analysing regexes with a Java implementation of the analyses. This Java implementation can be found in the software repository at [35]. All experiments were performed on a machine with hardware specifications as listed in Table 10.1.

10.2 Simple Analysis Results

The simple analysis was performed on two repositories of regexes, the Snort rule-set version 2.9.31 [36] containing 12499 regexes and RegExLib [37], containing 2994 regexes. As explained in Section 8.2.1, simple analysis provides an upper bound for the worst-case matching time of a regex, by calculating the degree of ambiguity of a related NFA. In practice, simple analysis was implemented to test for an upper bound of exponential matching time first, achieved by performing an algorithm to detect EDA. In the absence of an exponential upper bound, it proceeds to test for an upper bound of nonlinear matching time, by performing an algorithm that detects IDA. If such a nonlinear upper bound is found, the degree of IDA is also calculated. Therefore, performing

Table 10.1: Hardware specifications of the machine on which the experiments were performed.

Processor speed	3.1GHz
Number of cores	4
Cache size	6144Kb

Table 10.2: A breakdown of the simple analysis results.

Repository	Exp	Nonlin	Lin	Skipped	Timeout EDA	Timeout IDA
Snort	11	829	8385	3110	96	68
RegExLib	141	219	1606	973	13	42

Table 10.3: The degree of nonlinear growth for the regexes found to have a nonlinear upper bound for their worst-case matching time.

	Degree							
Repository	2	3	4	5	6	7	8	9
Snort	246	72	500	5	3	2	1	0
RegExLib	156	38	16	3	4	1	0	1

simple analysis experimentally on a regex yields one of six possible results. If the analysis completed successfully, the result will either be: Exponential (indicated with Exp); Nonlinear (Nonlin) or Linear (Lin). These results indicate the upper bound calculated for the worst-case matching time. If simple analysis failed to analyse the regex successfully, the results can be: Skipped, indicating a regex containing illegal syntax, requiring unhandled functionality or requiring an excessive amount of memory to complete; or analysis time exceeded some timeout value while calculating the degree of ambiguity (indicated as "Timeout EDA", or "Timeout IDA", depending on which calculation was running when the timeout occurred). Note that the simple analysis does not yield the result "Nonlinear" if the result "Exponential" would also be appropriate. A timeout of 10 minutes was used for each regex analysed with simple analysis. Running the simple analysis on the Snort and RegExLib repositories, yielded the results tabulated in Table 10.2. For the regexes found to have a nonlinear upper bound, the rate of growth was also calculated and is shown in Table 10.3.

From the results in Table 10.2 we can see that there are multiple regexes with an upper bound of exponential or nonlinear matching time. For many regexes, however, the simple analysis timed out. This happened in spite of the simple analysis having a polynomial worst-case execution time. In most of these cases, the long execution time of the analysis was caused by large, finite lower and upper bounds used in interval quantifiers. Our implementation of the analyses converts such interval quantifiers into pNFAs by repeating the operand subexpression to satisfy the lower and upper bounds. Consequently, interval quantifiers with a large range between the lower and upper bound may yield pNFAs of size potentially exponentially larger than the regex.

10.3 Full Analysis Results

If simple analysis indicates a nonlinear upper bound for worst-case matching time, full analysis is performed to determine whether this upper bound is

Table 10.4: The matching time behaviour, as determined by full analysis, of the 152 and 1048 regexes shown to have EDA and IDA, respectively, by simple analysis.

Simple Analysis	Full Analysis					
Exp	Exp	Nonlin	Lin	Skipped	Timeout EDA	Timeout IDA
152	137	0	2	1	12	0
Nonlin	Exp	Nonlin	Lin	Skipped	Timeout EDA	Timeout IDA
1048	0	394	24	3	0	627

Table 10.5: The degree of nonlinear growth for the regexes found to have nonlinear worst-case matching time.

	Degree			
Repository	2	3	4	5
Snort	202	13	1	0
RegExLib	118	25	6	1

tight. Strictly speaking, it is only necessary to perform full analysis if the exploit string, generated using the results of simple analysis, fails to trigger exponential (or nonlinear) matching time.

Simple analysis determined that, in total, 152 regexes have an upper bound of exponential worst-case matching time, 1048 have a nonlinear upper bound, and 9991 have neither. The remaining 4302 regexes were either skipped, or timed out. Full analysis was performed on the regexes with either an exponential or nonlinear upper bound for their worst-case matching time (as decided experimentally by simple analysis). The results of the full analysis are given in Table 10.4, which shows whether the worst-case matching time of a regex is exponential, nonlinear or linear; or whether the analysis required an exceeding amount of memory, or timed out. Again, the degree of nonlinear growth for all regexes determined to have nonlinear matching time, is shown in Table 10.5.

All regexes classified as having exponential worst-case matching time were tested against the Java regex matcher with their respective exploit strings. These exploit strings are constructed as explained in Chapter 9. The test involves repeatedly matching the vulnerable regex with the exploit string, increasing the length of the pump and measuring the matching time. If the matching time grows exponentially for a linear increase in the length of the pump, the regex has been proven to be vulnerable. All of the regexes classified to have exponential worst-case matching time did indeed exhibit this exponential growth in matching time and are therefore vulnerable.

10.4 Noteworthy Examples

To illustrate a case in which the simple analysis provides an inaccurate upper bound for the worst-case matching time of a regex, consider the regex `(\&d[0-9]{2}=.*?)+` from the Snort repository. This regex matches any input

string with a prefix `&d`, followed by two digits and, followed by `=`. In order to build an exploit string, we can use ε as prefix and `&d00=&d00=` as a pump. Since `.*?` matches all strings, two copies of the string `&d00=` can be matched in two ways: either once with the `\&d[0-9]{2}=` subexpression and once with the `.*?` subexpression, or twice with the `+` operator. In spite of this, every time the matcher cannot match part of the input string with the `(\&d[0-9]{2}=)` subexpression, it will backtrack and match one character with `.*?`, and thus all strings will be matched in linear time. In the simple analysis, it was detected that the regex has an upper bound of exponential worst-case matching time, due to exponential degree of ambiguity. However, when the full analysis is performed, it was detected that the states causing the EDA are unreachable when taking priorities into account. Therefore, full analysis classified the regex as not being vulnerable.

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

10.5 Conclusion

In this chapter we experimentally measured the viability of using static analysis to determine whether a regex is vulnerable. This was achieved by using an implementation of the theory developed in the previous chapters to analyse repositories of regexes used in practice. It was found that some regexes used in practice are indeed vulnerable. The vulnerability of these regexes was proven constructing an exploit string and experimentally measuring the matching time.

Chapter 11

Possible Solutions

11.1 Introduction

In this chapter we discuss how to turn vulnerable regexes into harmless equivalent regexes, in spite of the inherent vulnerability of backtracking matchers. As we have related regex ambiguity and regex vulnerability in Chapters 6 and 7, our techniques for fixing vulnerable regexes focus on reducing the degree of ambiguity of a regex and its subexpressions.

As we have seen in Chapter 7, not all unambiguous regexes are harmless. Specifically, recall the example given with the regex E_1E_2, where the subexpression E_1 has exponential degree of ambiguity and $\mathcal{L}(E_2) = \emptyset$. Therefore, when trying to fix a vulnerable regex, we rather need to reduce the degree of ambiguity of each of its subexpressions with infinite or exponential degree of ambiguity individually, rather the regex as a whole. A backtracking matcher is able to accept or reject any input string with minimal backtracking when matching with a regex containing no ambiguous subexpressions, and consequently, such regexes cannot be vulnerable.

The available techniques for the remediation of vulnerable regexes depend on the purpose of the regex in the incorporating software. If a vulnerable regex $E_{\mathtt{V}}$ is used solely for the purpose of membership testing to $\mathcal{L}(E_{\mathtt{V}})$, we may consider any harmless regex $E_{\mathtt{H}}$ such that $\mathcal{L}(E_{\mathtt{H}}) = \mathcal{L}(E_{\mathtt{V}})$, as a viable replacement option. On the other hand, if a regex is also used for parsing a successfully matched input string (such as explained in Chapter 3), our options for selecting an equivalent harmless regex are more restricted.

11.2 Equivalent Regex Replacement

When a vulnerable regex is used only to determine whether certain input strings are matched by it or not, such as a regex used for user input validation, then the only requirement on a replacement harmless regex is that it must accept the same input strings as the original vulnerable regex. We are therefore

looking for a harmless regex $E_{\mathtt{H}}$ such that $\mathcal{L}(E_{\mathtt{H}}) = \mathcal{L}(E_{\mathtt{V}})$, where $E_{\mathtt{V}}$ denotes the original vulnerable regex.

We start the process of obtaining $E_{\mathtt{H}}$ by finding unambiguous automata, and specifically DFA, that matches $\mathcal{L}(E_{\mathtt{V}}(i))$, where the $E_{\mathtt{V}}(i)$ are subexpressions of $E_{\mathtt{V}}$ with infinite or exponential degree of ambiguity.

It is well known that every NFA A can be converted to a (state) minimised DFA, denoted by $\mathrm{dfa}(A)$, recognising the same language, but with potentially exponentially many more states. One way of achieving this is to use the methods described in [38] and [39]. Thus for a vulnerable regex $E_{\mathtt{V}}$ with subexpressions $E_{\mathtt{V}}(i)$ having infinite or exponential degree of ambiguity, we can construct $E_{\mathtt{H}}$ by replacing $\mathrm{dfa}(\mathrm{nfa}(J^p(E_{\mathtt{V}}(i))))$ by language equivalent unambiguous expressions $E_{\mathtt{U}}(i)$.

In [29] a method, referred to as the flow-graph technique, is given for converting an NFA A into a language equivalent regex E. However, it is not guaranteed that after applying this method to a DFA, that a strongly unambiguous regex is obtained. We therefore give the adapted flow-graph technique from [40], for which this guarantee can indeed be made.

When applying the adapted flow-graph technique to a DFA $M = (Q = \{q_1, \ldots, q_n\}, \Sigma, q_1, \delta, F)$, let $R(i,j,k)$ be a regex for the language of strings obtained when starting and ending at states q_i and q_j respectively, and using only states $q_1, \ldots, q_k$ as intermediate states. We define $R(i,j,0)$ as:

$$R(i,j,0) = \begin{cases} (\alpha \mid \beta \mid \ldots) & \text{if we have transitions on } \alpha, \beta, \ldots \\ & \text{from } q_i \text{ to } q_j; \\ \emptyset & i \neq j, \text{ and we have no transitions} \\ & \text{from } q_i \text{ to } q_j; \\ \varepsilon & i = j, \text{ and we have no transitions} \\ & \text{from } q_i \text{ to } q_j. \end{cases}$$

For $i \neq k, j \neq k, k > 0$ let

$$R(i,j,k) = R_1 \mid R_2 {R_3}^* R_4,$$

where

$$\begin{aligned} R_1 &= R(i,j,k-1); \\ R_2 &= R(i,k,k-1); \\ R_3 &= R(k,k,k-1); \\ R_4 &= R(k,j,k-1). \end{aligned}$$

In order to obtain a strongly unambiguous regex (with no subexpression E such that $\mathcal{L}(E) = \emptyset$), we simplify $R(i,j,k)$, for $i \neq k, j \neq k, k > 0$, as follows. If $R_2 = \emptyset$ or $R_4 = \emptyset$ we let

$$R(i,j,k) = R_1,$$

and if $R_3 = \varepsilon$, we instead let

$$R(i,j,k) = R_1 \mid R_2 R_4.$$

For $i = k$, $j \neq k$ let

$$R(i, j, k) = {R_1}^* R_2,$$

where

$$\begin{aligned} R_1 &= R(k, k, k-1); \\ R_2 &= R(k, j, k-1), \end{aligned}$$

with the simplification that if $R_1 = \varepsilon$, then

$$R(i, j, k) = R_2.$$

Similarly, when $i \neq k$, $j = k$

$$R(i, j, k) = R_1 {R_2}^*,$$

where

$$\begin{aligned} R_1 &= R(i, k, k-1); \\ R_2 &= R(k, k, k-1), \end{aligned}$$

except for $R_2 = \varepsilon$, when we simplify as follows:

$$R(i, j, k) = R_1.$$

Finally, when $i = j = k$

$$R(i, j, k) = {R_1}^*,$$

where

$$R_1 = R(i, j, k-1),$$

except for $R_1 = \varepsilon$, when we simplify as follows:

$$R(i, j, k) = \varepsilon.$$

The regex equivalent to the original DFA is now obtained as

$$R(1, j_1, n) \,|\, \ldots \,|\, R(1, j_m, n)\,,$$

where $\{q_{j_1}, \ldots, q_{j_m}\} = F$.

This adapted flow-graph technique produces a strongly unambiguous regex, since it ensures a one-to-one correspondence between paths in the original DFA and paths in the constructed regex.

We illustrate the usage of the adapted flow-graph technique by finding a harmless regex equivalent to the vulnerable regex $E_{\mathtt{V}} = (\mathtt{\backslash w^+(/\backslash w^+.^{*?})^*[.]txt})$, used for matching paths in a directory structure to text files. The regex $E_{\mathtt{V}}$ contains a subexpression $E_{\mathtt{V}}(5) = (\mathtt{/\backslash w^+.^{*?}})^*$ with exponential degree of ambiguity. To see that this subexpression has exponential ambiguity, note that the annotated regex $Y_{\mathrm{sA}}(E_{\mathtt{V}}(5))$ equals $1(23/45(6\mathtt{\backslash w})^+7(8.)^{*?})^*$, and observe that $Y_{\mathrm{sA}}(E_{\mathtt{V}}(5))$ matches, among others, strings of the form $1s_0s_1 \ldots s_n$, where $s_i \in \{23/456\mathtt{a}723/456\mathtt{a}7, 23/456\mathtt{a}78/8\mathtt{a}\}$. Since $\pi_\Sigma(1s_0s_1 \ldots s_n) = (\mathtt{/a/a})^n$ in

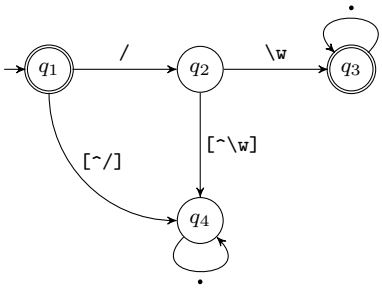

Figure 11.1: The minimised DFA dfa(nfa(J^p((/\w$^+$.$^{*?}$)*))).

Table 11.1: The values of $R(i,j,k)$ when applying the flow-graph technique to dfa(nfa(J^p((/\w$^+$.$^{*?}$)*))).

$R(i,j,0)$

i \ j	0	1	2	3
0	ε	/	$\emptyset$	[^/]
1	$\emptyset$	ε	\w	[^\w]
2	$\emptyset$	$\emptyset$	.	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	.

$R(i,j,1)$

i \ j	0	1	2	3
0	ε	/	/\w	[^/]|/[^\w]
1	$\emptyset$	ε	\w	[^\w]
2	$\emptyset$	$\emptyset$	.	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	.

$R(i,j,2)$

i \ j	0	1	2	3
0	ε	/	/\w.*	[^/]|/[^\w]
1	$\emptyset$	ε	\w.*	[^\w]
2	$\emptyset$	$\emptyset$	.*	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	.

$R(i,j,3)$

i \ j	0	1	2	3
0	ε	/	/\w.*	[^/]|/[^\w].*
1	$\emptyset$	ε	\w.*	[^\w].*
2	$\emptyset$	$\emptyset$	.*	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	.

each case, there are 2^n words $v_i \in \mathcal{L}(Y_{\text{sA}}(E_{\texttt{V}}(5)))$ such that $\pi_\Sigma(v_i) = w$ for $w = (\texttt{/a/a})^n$, and thus $E_{\texttt{V}}(5)$ has exponential degree of ambiguity.

Having determined the subexpression causing the vulnerability in $E_{\texttt{V}}$, we need to find a suitable alternative. We construct $A = \text{dfa}(\text{nfa}(J^p(E_{\texttt{V}}(5))))$, as shown in Figure 11.1. Intuitively, we can confirm the correctness of A by noting that $E_{\texttt{V}}(5)$ will match either the empty string, or any input string with the prefix $/w$, where w is a word character. Next, we apply the flow-graph technique to A. The different values of $R(i,j,k)$, required for this process is shown in Table 11.1. Using the values from the table, we obtain the strongly unambiguous regex $E_{\texttt{U}}(5)$ matching $\mathcal{L}(E_{\texttt{V}}(5))$, given by $(R(0,0,3) \mid R(0,2,3)) = (\varepsilon \mid \texttt{/\w.}^*)$. Finally, we obtain $E_{\texttt{H}}$ by substituting $E_{\texttt{V}}(5)$ by $E_{\texttt{U}}(5)$, which yields $E_{\texttt{H}} = (\texttt{\w}(\varepsilon \mid \texttt{/\w.}^*)\texttt{[.]txt})$.

In general, one cannot simply replace a vulnerable regex by any harmless language equivalent regex, as this might change the behaviour of the capturing groups (or even remove them). We illustrate this with the previous example where $E_{\mathtt{V}} = (\mathtt{\backslash w}^{+}(\mathtt{/\backslash w}^{+}\mathtt{.}^{*?})^{*}\mathtt{[.]txt})$. In $E_{\mathtt{V}}$, the capturing group $(\mathtt{/\backslash w}^{+}\mathtt{.}^{*?})$ can be used to capture the filename in the path to the text file matched by the regex, but in $E_{\mathtt{H}} = (\mathtt{\backslash w}(\varepsilon \mid \mathtt{/\backslash w.}^{*})\mathtt{[.]txt})$, however, the capturing group $(\varepsilon \mid \mathtt{/\backslash w.}^{*})$ will also match the names of directories in the path as well.

11.3 Using Atomic Groups and Possessive Quantifiers

As explained in Section 3.4.5, atomic groups and possessive quantifiers instruct a regex matcher to disregard certain matching options. The concept of possessive quantifiers is described in [41] as follows:

> *"A feature I think would be useful, but that no regex flavor that I know of has, is what I would call possessive quantifiers. They would act like normal quantifiers except that once they made a decision that met with local success, they would never backtrack to try the other option. The text they match could be unmatched if their enclosing subexpression was unmatched, but they would never give up matched text of their own volition, even in deference to an overall match."*

Subsequent to this quote, both possessive quantifiers and a generalisation thereof, atomic groups, were added to some regular expression matcher implementations, such as in Java. As possessive quantifiers can be represented with atomic groups, our discussion will focus only on the latter.

Atomic groups can be used to remove vulnerabilities in some regexes, by instructing the matcher to disregard the backtracking possibilities discovered within the atomic group. For example, the regex $(E_1 \mid E_2)^*$, for unambiguous expressions E_1 and E_2 where $w \in \mathcal{L}(E_1) \cap \mathcal{L}(E_2)$, where $w \neq \varepsilon$, has exponential degree of ambiguity. In this example, strings of the form $w^n r$, where $r \notin (E_1 \mid E_2)^*$, can be used as exploit strings. Using the atomic operator (or making the Kleene star possessive) to change $(E_1 \mid E_2)^*$ into $(\mathtt{?>}(E_1 \mid E_2)^*)$ or $(\mathtt{?>}(E_1 \mid E_2))^*$, will remove the vulnerability, due to the fact that backtracking possibilities are forgotten by the matcher after matching a substring (of an exploit string) with an atomic group.

Unfortunately, the above mentioned strategy cannot be used to remove vulnerabilities from all regexes. In some cases, introducing an atomic group in a regex changes the language accepted by the regex. Consider for example the vulnerable regex $((\mathtt{a} \mid \mathtt{b})^*)^*\mathtt{b}$, which can be exploited by using strings of the form $\mathtt{a}^n\mathtt{c}$. Modifying the regex to $(\mathtt{?>}((\mathtt{a} \mid \mathtt{b})^*)^*)\mathtt{b}$ or $(\mathtt{?>}(\mathtt{a} \mid \mathtt{b})^*)^*\mathtt{b}$, will have the effect that the strings $\mathtt{a}^k\mathtt{b}$ are no longer matched, while the regexes $(\mathtt{?>}((\mathtt{a} \mid \mathtt{b})^*)^*\mathtt{b})$ and $((\mathtt{?>a} \mid \mathtt{b})^*)^*\mathtt{b}$ are still vulnerable to the exploit strings $\mathtt{a}^n\mathtt{c}$.

A more in-depth discussion on how to model regexes with atomic groups can be found in [42].

11.3.1 Examples

Before we consider an example of how atomic groups can be used to remove a vulnerability from a regex, let us first consider a regex for which this is not possible. Consider the regex R_1 used to validate the path to an HTML file, extracted from the RegExLib repository [37], given by:
`^(([a-zA-Z]:)|(\\{2}\w+)\$?)(\\(\w[\w ]*.*))+\.((html|HTML)|(htm|HTM))$`
R_1 is vulnerable due to the exponential ambiguity in the subexpression $S_1 =$ `(\\(\w[\w ]*.*))+`. Note that the string `\a\a` can be matched either using the subexpressions $E_1 =$ `\\`, $E_2 =$ `\w[\w ]*` and $E_3 =$ `.*`, or using only E_1 and E_2 and using twice the outer `+` operator. Changing the subgroup S_1 to be atomic, i.e., changing S_1 to $A_1 =$ `(?>(\\(\w[\w ]*.*))+)`, would indeed remove the vulnerability, but since the atomic subgroup contains a greedy Kleene star subexpression, A_1 will match the rest of the input string, hogging the symbols from the suffix subexpression `\.((html|HTML)|(htm|HTM))$`. Therefore, this regex cannot be fixed, at least not in the obvious ways, by using atomic groups. Fortunately, the vulnerability can be removed without using atomic operators. Inspecting subgroup S_1, it can be seen that it accepts any string with the prefix `\a`. Consequently, we can substitute S_1 with the subgroup `(\\(\w.*))`. Applying this fix does not change the behaviour of the capturing groups. Note that for any matching input string the `+` operator in the subgroup S_1 is superfluous. The `.*` subexpression will consume the rest of the input string and symbols will be released until they are rematched with the `\.((html|HTML)|(htm|HTM))$` subexpression. Furthermore, `[\w ]*.*` is equivalent to `.*`. Therefore, the subexpression `(\w.*)` will match in the same way as the subexpression `(\w[\w ]*.*)`, and so the capturing of these two groups will be equivalent. The same goes for the capturing groups in the subexpressions `(\\(\w[\w ]*.*))` and `(\\(\w.*))`.

Next we consider a regex, referred to as R_2, which is a slightly simplified version of a regex extracted from the RegExLib [37] repository,
`^([0-9a-z]([-.\w]*[0-9a-z])*@(([0-9a-z])+([-\w]*[0-9a-z])*\.)+[a-z]{2,9})$`
used for email address validation. This regex is vulnerable due to the subgroups $S_{1,1} =$ `([-.\w]*[0-9a-z])*` and $S_{1,2} =$ `([-\w]*[0-9a-z])*`. Both $S_{1,1}$ and $S_{1,2}$ have exponential degree of ambiguity and can match the input string $(\texttt{aa})^n$ in exponentially many ways, in n. Note that $S_{1,1}$ is followed by the subexpression `@`, which does not match any common strings also matched by $S_{1,1}$. Consequently, modifying $S_{1,1}$ to be atomic, cannot hog any symbols from the suffix expressions. Therefore, $S_{1,1}$ can be made atomic and it will no longer contribute to the vulnerability of the regex. A similar argument holds for why $S_{1,2}$ can be made atomic. Therefore, the vulnerability of this email address validation regex can be fixed by changing the subgroups $S_{1,1}$ and

$S_{1,2}$ to be atomic, that is `(?>([-.\w]*[0-9a-z])*)` and `(?>([-\w]*[0-9a-z])*)`, respectively.

11.4 Conclusion

In this chapter, we have investigated the problem of mitigating the vulnerability of evil regexes. Two methods for achieving this have been provided and the context in which each method would be preferable has been investigated.

Chapter 12

Future Work and Conclusions

12.1 Future Work

12.1.1 Extending the Regex Mathematical Model

An obvious route for future work is to extend the analyses Defined in Chapter 8 to accommodate more of the features found in modern regex matchers. Some of the features not supported in our current model includes atomic groups, lookaround assertions, and backreferences. For a description of each of these features, refer to Chapter 3. In this section we briefly discuss mathematical models that might have the potential to model these features.

12.1.2 Modeling with Two-way Pushdown Automata

To model the behaviour of atomic groups in a regex, one might consider using *a two-way pushdown automaton (2DPDA)* [43]. A 2DPDA is a finite-state automaton with a stack that allows for the storage of stack symbols. The 2DPDA deterministically selects the next state to transition to, depending on the symbol at the current position in the input string and the current symbol at the top of the stack. With each transition of a 2DPDA, in addition to the next state being chosen, a symbol may be pushed onto or popped from the stack, and the position in the input string may be moved to the left or right.

Using a 2DPDA to model the matching behaviour of a backtracking regex matcher will allow backtracking possibilities to be stored as states on the stack. This type of 2DPDA will have two modes, namely a *matching* and a *backtracking mode* (these modes can be encoded into the states of the automaton).

In matching mode, when a 2DPDA reads a symbol from the input string, similar to an NFA, the position in the input string is moved to the right and the automaton transitions to a new state. The 2DPDA pushes a symbol onto the stack for every input symbol read, say X. Priorities are modeled by the 2DPDA, by always traversing to the state corresponding to the highest priority

choice (making it deterministic) and pushing the lower priority options, in order, onto the stack. These states can be transitioned to when backtracking.

In backtracking mode, the 2DPDA pops X symbols from the stack while moving left in the input string, until a state is found at the top of the stack. This state is then popped from the stack and transitioned to.

A 2DPDA has the added benefit of being able to model atomic groups more intuitively. Special symbols are pushed onto the stack when entering and exiting the atomic group, say A_i and Z_i, where i is be used to distinguish between specific atomic groups. When the 2DPDA is in backtracking mode and a Z_i symbol is found at the top of the stack (indicating the exit of an atomic group), the 2DPDA pops symbols from the stack (including stored, lower priority states) until the corresponding A_i symbol is found. After this, it resumes the regular backtracking behaviour and transitions to the next state found on the stack. This will effectively eliminate all backtracking possibilities originating from within the atomic group.

As 2DPDAs accept languages that are not regular [43], one cannot simply convert such an automaton to a pNFA and use the analyses developed in Chapter 8 to detect vulnerable regexes. Using these automata to detect vulnerable regexes would require an analysis procedure to be developed specifically for 2DPDAs.

12.1.3 Modeling with Alternating Finite-state Automata

A model that might be promising for modeling regexes with lookaround assertions, specifically positive lookahead assertions, is called an *alternating finite-state automaton (AFA)* [44]. AFAs are a generalisation of NFAs. Where an NFA will accept an input string if any nondeterministic path ends in an accept state, an AFA can require that multiple paths end in an accept state. This is achieved by defining both universal states ($\forall$-states) and existential states ($\exists$-states). To explain the intuition of matching with an AFA, we will explain how it differs to matching with an NFA. For an NFA, when a nondeterministic choice is encountered when reading the input string, it can be thought of as if the NFA branches into multiple NFAs being run in parallel. Each parallel NFA transitions to a different state presented within the nondeterministic choice. The NFA accepts the input string if any of these parallel NFAs accept the input string. The $\exists$-states of an AFA exhibit the same behaviour. Indeed, an AFA containing only $\exists$-states, is equivalent to an NFA. Conversely, for the $\forall$-states, the AFA branches into multiple AFAs run in parallel and the input string is only accepted if *every* parallel AFA, generated at this state, accepts the input string.

The $\forall$-states in an AFA are useful for modeling positive lookahead assertions in regexes. A positive lookahead assertion requires that the expression

contained therein matches a prefix of the remainder of the input string. Therefore, a successful match is only possible if this requirement is fulfilled, as well as a successful match with all subsequent subexpressions. A $\forall$-state can be used to enforce both these requirements in a regex.

As the sets of languages accepted by AFAs are exactly the set of regular languages, it is possible to construct an NFA matching the same language as an AFA [44]. Ideally, an NFA converted from an AFA should then be used to analyse regexes with positive lookahead assertions. It was found, however, that the conversion from AFA to NFA causes inaccuracies in the analysis. As explained in Chapter 3, the Java regex matcher handles positive lookahead assertions by first matching with the expression contained within the assertion. If this match is successful, it resets its position in the input string to where it was before matching with the assertion expression. It then proceeds to match with the subexpressions subsequent to the positive lookahead assertion. This causes the disparities between the matching time observed in the Java regex matcher and the results of analysing an NFA converted from an AFA. Consider for example a regex $(\texttt{?=}E_1)E_2$, where $E_1\texttt{.}^*$ is vulnerable when using exploit strings of the form $w^n r$, and assume $\mathcal{L}(E_1) \cap \mathcal{L}(E_2) = \emptyset$. Using $(\texttt{?=}E_1)E_2$ to match $w^n r$, will still exhibit exponential matching time in n, as the matcher first attempts to match the exploit string with the vulnerable subexpression. However, removing the positive lookahead assertion by converting an AFA constructed for $(\texttt{?=}E_1)E_2$ to an NFA, will also remove the vulnerability. It is simple to confirm this, as this NFA will match $\emptyset$.

12.1.4 Modeling with Memory Automata

Finally, when attempting to analyse regexes with backreferences, one could look at *memory automata (MFA)* [3; 45]. An MFA allows for the storage of substrings of the input string matched by the matcher, in memory cells. These substrings can then be recalled and compared to the remainder of the input string. This can be used to model backreferences, by creating a memory cell for every capturing that is backreferenced to in the regex. When matching with the subexpression within the capturing group, the substring can be stored within the memory cell. When the backreference subexpression is encountered, this memory cell is then consulted for the substring that was matched. As MFA are capable of matching languages that are not regular, one would have to develop new analysis techniques to decide whether the regex modeled with an MFA is vulnerable.

12.2 Conclusions

ReDoS presents a formidable security threat to web applications that rely on regexes, as illustrated by examples given of ReDoS that occurred in practice.

However, in spite of these occurrences, the phenomenon remains largely unexplored from a theoretical perspective.

By using formal language theory as a basis and performing an inspection of the source code of backtracking regex matchers it was possible to derive a concise mathematical model of regex backtracking matching behaviour in a modern regex matcher. The model is defined in terms of pNFAs (prioritised nondeterministic finite-state automata) and is capable of modeling both the traditional features of regexes, as well as some of the extensions implemented in modern regex matchers.

We used pNFAs in conjunction with the notion of ambiguity of nondeterministic finite-state automata, as well as the concepts of weak and strong ambiguity of regexes, to define a vulnerable regex formally. The accuracy of this definition was investigated as well as which exceptions could exist therefor. We then derived a procedure to detect vulnerabilities in regexes. Our techniques were validated by testing it on two repositories of regexes used in practice. It was found that some regexes used in practice are indeed vulnerable.

If a regex is classified as vulnerable there must exist some input string that can trigger inordinately long matching time. We explained a procedure to construct such input strings. Using this procedure, we confirmed the vulnerability of all regexes classified as having worst-case exponential matching time by our techniques by experimentally measuring the matching time when using the Java regex matcher. We have suggested approaches which can be used to mitigate the vulnerabilities after they have been found, where the best approach depends on the context in which the regex is used in the software. Lastly, we have investigated multiple, potential avenues for extending our analysis of regexes by providing automata theoretical models which could be used to model some of the regex extensions implemented in modern matchers.

In conclusion, the research presented within this thesis provides a basis for software developers and academics to understand the concept of evil regexes. In turn, this will help raise awareness of the problem and provide the computer science community with the means to defend against ReDoS attacks.

Appendices

Appendix A

Nonprecise Modeling of Repeated Epsilon Transitions

Before we can explain the cases in which ε-transitions may be repeated, we need to understand in greater detail how capturing groups operate. Let us emphasise capturing groups in regexes by adding integer subscripts to the opening and closing parenthesis of a capturing group. For example, in the regex $E({}_1F)_1R$, the capturing group, group 1, is denoted by the subscripts 1 after the parentheses. After an input string is successfully matched with a regex, each capturing group contains the substring that was matched by the corresponding subexpression. Continuing with the regex of our previous example, $E({}_1F)_1R$, suppose $w \in \mathcal{L}(E({}_1F)_1R)$. After matching this regex with w, the capturing group, group 1, will contain the substring $w_1 \in \mathcal{L}(F)$ of w, matched with the subexpression F.

When a capturing group is used multiple times in a successful matching process, it contains only the last substring matched by the expression within the capturing group. For example, consider the regex $(({}_1[\texttt{ab}])_1\texttt{c})^*$. After matching the input string `acacbc`, group 1 will contain the substring `b`, although it also matched an `a` during the matching process. If a capturing group is bypassed entirely during the matching process, such as when matching the input string `b` with the regex $({}_1\texttt{a})_1 \mid \texttt{b}$, we say the capturing group is NULL. Note that this should not be confused with the case in which the capturing group is traversed, but matches the empty string.

Finally, consider the pNFA shown in Figure A.1, which represents the regex $({}_1a^*)_1{}^*$. The transition $q_0 \to q_1$ opens up the capturing group and $q_1 \to q_0$ closes it. Any symbols read between the last traversal of these two transitions will be contained in the capturing group after the match has completed (if no symbols are read, the capturing group will contain ε). When matching the input string `aaa` with this regex using Java, the capturing group contains the substring ε. Thus after `aaa` was matched, the matcher proceeds to match the empty string with $({}_1a^*)_1$. Looking at the pNFA in Figure A.1, this suggests that after leaving the inner $({}_1a^*)_1$ via the $q_1 \to q_0$ transition, the matcher again

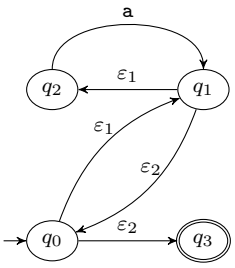

Figure A.1: (a) A pNFA simulating the matching behaviour of the regex $(\mathtt{a}^*)^*$ in Java.

takes the $q_0 \to q_1 \to q_0$ ε-loop to accept the empty string. This implies that the $q_1 \to q_0$ transition was traversed twice without consuming an intermediate input symbol.

To model the behaviour of the Java backtracking matcher more accurately, one would have to develop a model which allows for the reuse of certain ε-transitions, without consuming an input symbol between these uses. In spite of this, as the model used currently is sufficient for the detection of vulnerable regexes (as shown in Chapter 10), it was decided to use the simpler, albeit less accurate model.

Appendix B

Sensitive Data Exposure and Evil Regexes

In some contrived situations, evil regexes can be exploited to expose sensitive data. The circumstances that may lead to this vulnerability are listed below.

1. A user must be able to specify a regex to be used by the system.
2. This regex must be matched against sensitive data.
3. The regex matcher used must be vulnerable to evil regexes.

Note that since the user-controlled regex is matching sensitive data, we may assume that the user is not shown the result of this matching, otherwise exposing the sensitive data would be trivial.

As an example, suppose that a user-controlled regex is matching a 32 character hexadecimal password hash. The core of the exploit rests on the fact that the regex `.{32}` will successfully match the password hash in much less time than a regex such as `(.*){`N`}G` would, for some large value N, due to IDA. Ideally, N should be chosen large enough to create a noticeable delay in the response time of the system, but not so large as to cause it to enter a state of denial of service. The `G` in the regex is simply used to force a match with the hexadecimal characters of the password hash to fail. In the general case, the expression `[A&&B]` can be used to force a match to fail (this expression is the character class formed from the result of the intersection of the character classes `[A]` and `[B]`, which is empty).

Giving the regex `(([0-7].{31})|((.*){100}G))` to the system, will allow us to infer information about the first hexadecimal character of the password hash. Inspecting this regex reveals that the matching time will be fast for all password hashes starting with a character in the character class `[0-7]`. For these password hashes, the match will complete in linear time with the `([0-7].{31})` branch of the alternation. All other password hashes will reject the `[0-7]` character class and rather match with the `((.*){100}G))` branch,

for which the matching time will be polynomial of high degree (supposing, however, that the matching time is still short enough to wait for the system to complete in a reasonable amount of time). Therefore, timing the response time of the system will allow us to infer whether the first hexadecimal character of the password hash is the character class `[0-7]`, or otherwise in `[89A-F]`.

After the first inference, we can tighten the bounds of the character class we now know the first character to be in. Suppose, without loss of generality, that the character is in the character class `[0-7]`. We now give the system the regex `(([0-3].{31})|((.*){100}G))`. Timing the response time will tell us whether the character is within the character class `[0-3]`, or otherwise in `[4-7]`. By continuing in this fashion of measuring the response time and tightening the bounds of the character classes, we can calculate the first character of the password hash exactly. After the first character has been calculated, we can infer the other characters by using the regexes `((.{1}[0-7].{30})|((.*){100}G))`, ..., and `((.{31}[0-7])|((.*){100}G))` in a similar manner.

This regex based binary search allows us to infer the password hash exactly with 128 or less regexes.

Even though the example given is contrived and may possibly never occur in practice, it is nevertheless important to know that evil regexes might be used for an attack on an aspect of information security other than availability.

List of References

[1] Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N. and Barzilay, R.: Neural generation of regular expressions from natural language with minimal domain knowledge. *CoRR*, vol. abs/1608.03000, 2016.

[2] Backurs, A. and Indyk, P.: Which regular expression patterns are hard to match? In: Dinur, I. (ed.), *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pp. 457–466. IEEE Computer Society, 2016. ISBN 978-1-5090-3933-3.

[3] Schmid, M.L.: Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, vol. 249, pp. 1–17, 2016.

[4] Regular expression static analysis project page. Accessed: April, 2016. Available at: `http://www.cs.sun.ac.za/%7Eabvdm/regex.html`

[5] Outage postmortem - july 20, 2016. Accessed: June, 2017. Available at: `http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016`

[6] Runaway Regular Expressions: Catastrophic Backtracking. Accessed: October, 2016. Available at: `http://www.regular-expressions.info/catastrophic.html`

[7] Aho, A.V., Kernighan, B.W. and Weinberger, P.J.: *The AWK Programming Language*. Addison-Wesley, 1988.

[8] MySQL. Accessed: October, 2016. Available at: `http://www.mysql.com/`

[9] RE2. Accessed: March, 2017. Available at: `https://github.com/google/re2`

[10] Perl Compatible Regular Expressions. Accessed: November, 2017. Available at: `https://www.pcre.org/`

[11] Sipser, M.: *Introduction to the Theory of Computation*. 3rd edn. International Thomson Publishing, 2013. ISBN 1133187811.

[12] Berglund, M. and van der Merwe, B.: On the semantics of regular expression parsing in the wild. In: Drewes, F. (ed.), *20th Intl. Conf. on Implementation and Application of Automata (CIAA 2015)*, vol. 9223 of *LNCS*, pp. 292–304. 2015. ISBN 978-3-319-22359-9.

[13] Weber, A. and Seidl, H.: On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, vol. 88, no. 2, pp. 325–349, 1991.

[14] Allauzen, C., Mohri, M. and Rastogi, A.: General algorithms for testing the ambiguity of finite automata. In: Ito, M. and Toyama, M. (eds.), *Proc. 12th Intl. Conf. on Developments in Language Theory (DLT 2008)*, vol. 5257 of *LNCS*, pp. 108–120. 2008. ISBN 978-3-540-85779-2.

[15] Friedl, J.E.F.: *Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.)*. O'Reilly, 2006. ISBN 978-0-596-52812-6.

[16] Weideman, N., van der Merwe, B., Berglund, M. and Watson, B.: Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In: Han, Y. and Salomaa, K. (eds.), *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, vol. 9705 of *Lecture Notes in Computer Science*, pp. 322–334. Springer, 2016. ISBN 978-3-319-40945-0.

[17] Ben-Porat, U., Bremler-Barr, A. and Levy, H.: Vulnerability of network mechanisms to sophisticated ddos attacks. *IEEE Trans. Computers*, vol. 62, no. 5, pp. 1031–1043, 2013.

[18] Crosby, S.A. and Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[19] Smith, R., Estan, C. and Jha, S.: Backtracking algorithmic complexity attacks against a NIDS. In: *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pp. 89–98. IEEE Computer Society, 2006. ISBN 0-7695-2716-7.

[20] The Open Web Application Security Project. Accessed: October, 2016.
Available at: `https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS`

[21] Roichman, A. and Weidman, A.: Regular expression denial of service. 2012. Accessed: October, 2016.
Available at: `https://www.checkmarx.com/white_papers/redos-regular-expression-denial-of-service/`

[22] Cox, R.: Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). 2007. Accessed: October, 2016.
Available at: `https://swtch.com/%7Ersc/regexp/regexp1.html`

[23] Berglund, M., Drewes, F. and van der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching. In: Ésik, Z. and Fülöp, Z. (eds.), *Proce. 14th Intl. Conf. on Automata and Formal Languages (AFL 2014)*, vol. 151 of *EPTCS*, pp. 109–123. 2014.

[24] Kirrage, J., Rathnayake, A. and Thielecke, H.: Static analysis for regular expression denial-of-service attacks. In: Lopez, J., Huang, X. and Sandhu, R. (eds.), *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, vol. 7873 of *Lecture Notes in Computer Science*, pp. 135–148. Springer, 2013. ISBN 978-3-642-38630-5.

[25] Rathnayake, A. and Thielecke, H.: Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, vol. abs/1405.7058, 2014.

[26] Drewes, F., van der Merwe, B. and Weideman, N.: The output size problem for string-to-tree transducers. In: *Workshop on Trends in Tree Automata and Tree Transducers (TTATT 2016)*. 2016.

[27] Van der Merwe, B., Drewes, F. and Berglund, M.: The output size problem for string-to-tree transducers. 2017. Submission to the *Journal of Automata, Languages and Combinatorics*.

[28] Sippu, S. and Soisalon-Soininen, E.: Parsing theory, vol. i: Languages and parsing, volume 15 of eatcs monographs on theoretical computer science. 1988.

[29] Book, R., Even, S., Greibach, S. and Ott, G.: Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 149–153, 1971.

[30] Brüggemann-Klein, A.: Regular expressions into finite automata. *Theoretical Computer Science*, vol. 120, no. 2, pp. 197–213, 1993.

[31] Glushkov, V.M.: The abstract theory of automata. *Russian Mathematical Surveys*, vol. 16, no. 5, p. 1, 1961.

[32] Thompson, K.: Regular expression search algorithm. *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[33] Meyer, A.R. and Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: *13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972*, pp. 125–129. IEEE Computer Society, 1972.

[34] Wüstholz, V., Olivo, O., Heule, M.J.H. and Dillig, I.: Static detection of dos vulnerabilities in programs that use regular expressions (extended version). *CoRR*, vol. abs/1701.04045, 2017.

[35] Regex Static Analysis. Accessed: March, 2017.
Available at: `https://github.com/NicolaasWeideman/RegexStaticAnalysis`

[36] Snort. Accessed: October, 2015.
Available at: `http://www.snort.org`

[37] RegexLib. Accessed: October, 2015.
Available at: `http://www.regxlib.com`

[38] Rabin, M.O. and Scott, D.S.: Finite automata and their decision problems. *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, 1959.
Available at: `https://doi.org/10.1147/rd.32.0114`

[39] Hopcroft, J.: An $n \log n$ algorithm for minimizing states infinite automaton. *Technical report*, 1971.

[40] Van der Merwe, B., Weideman, N. and Berglund, M.: Turning evil regexes harmless. In: *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, pp. 38:1–38:10. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-5250-5.

[41] Friedl, J.E.F.: *Mastering regular expressions - powerful techniques for Perl and other tools*. Powerful techniques for Perl and other tools. O'Reilly, 1997. ISBN 978-1-56592-257-0.

[42] Berglund, M., van der Merwe, B., Watson, B. and Weideman, N.: On the semantics of atomic subgroups in practical regular expressions. In: Carayol, A. and Nicaud, C. (eds.), *Implementation and Application of Automata - 22nd International Conference, CIAA 2017, Marne-la-Vallée, France, June 27-30, 2017, Proceedings*, vol. 10329 of *Lecture Notes in Computer Science*, pp. 14–26. Springer, 2017. ISBN 978-3-319-60133-5.

[43] Gray, J., Harrison, M.A. and Ibarra, O.H.: Two-way pushdown automata. *Information and Control*, vol. 11, no. 1/2, pp. 30–70, 1967.

[44] Chandra, A.K., Kozen, D. and Stockmeyer, L.J.: Alternation. *J. ACM*, vol. 28, no. 1, pp. 114–133, 1981.

[45] Freydenberger, D.D. and Schmid, M.L.: Deterministic regular expressions with back-references. In: Vollmer, H. and Vallée, B. (eds.), *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, vol. 66 of *LIPIcs*, pp. 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. ISBN 978-3-95977-028-6.
Available at: `https://doi.org/10.4230/LIPIcs.STACS.2017.33`