Theory and Patterns for Avoiding Regex Denial of Service

Sk Adnan Hassan

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science & Applications

Francisco Javier Servant Cortes, Chair

Na Meng

Muhammad Ali Gulzar

May 4, 2022

Blacksburg, Virginia

Keywords: security, denial of service, redos

Copyright 2022, Sk Adnan Hassan

Theory and Patterns for Avoiding Regex Denial of Service

Sk Adnan Hassan

(ABSTRACT)

Regular expressions are ubiquitous. They are used for diverse purposes, including input validation and firewalls. Unfortunately, they can also lead to a security vulnerability called ReDoS(Regular Expression Denial of Service), caused by a super-linear worst-case execution time during regex matching. ReDoS has a serious and wide impact: since applications written in most programming languages can be vulnerable to it, ReDoS has caused outages at prominent web services including Cloudflare and Stack Overflow. Due to the severity and prevalence of ReDoS, past work proposed mechanisms to identify and repair regexes. In this work, we set a different goal: helping developers avoid introducing regexes that could trigger ReDoS in the first place. A necessary condition for a regex to trigger ReDoS is to be infinitely ambiguous (IA). We propose a theory and a collection of anti-patterns to characterize infinitely ambiguous (IA) regexes. We evaluate our proposed anti-patterns in two complementary ways: quantitatively, over a dataset of 209,188 regexes from open-source software; and qualitatively, by observing humans using them in practice. In our large-scale evaluation, our anti-patterns characterized IA regexes with 100% precision and 99% recall, showing that they can capture the large majority of IA regexes, even when they are a simplified version of our theory. In our human experiment, practitioners applying our anti-patterns correctly assessed whether the regex that they were composing was IA or not in all of our studied regex-composition tasks.

Theory and Patterns for Avoiding Regex Denial of Service

Sk Adnan Hassan

(GENERAL AUDIENCE ABSTRACT)

Regular expressions used by developers for different purposes, including input validation and firewalls. Unfortunately, they can also lead to a security vulnerability called ReDoS(Regular Expression Denial of Service), caused by a super-linear worst-case execution time during regex matching. ReDoS has caused outages at prominent web services including Cloudflare and Stack Overflow. ReDoS has a serious and wide impact: since applications written in most programming languages can be vulnerable to it. With this work, we wanted to help developers avoid introducing regexes that could trigger ReDoS in the first place. A necessary condition for a regex to trigger ReDoS is to be infinitely ambiguous (IA). We propose a theory and a collection of anti-patterns to characterize infinitely ambiguous (IA) regexes.

Dedication

To my lovely wife Tultul and to my family

Acknowledgments

I would like to thank my advisor, Dr. Francisco Servant, who guided me in every step of this journey. Thank you for all your support and feedback.

Big thanks to wife Tultul, you supported me in those difficult nights where no one was there.

To my family who gave me support in all my decisions throughout my life.

I would like to thank my collaborators Dr. James C. Davis and Dr. Dongyoon Lee for your guidance, feedback and support throughout this journey.

I would also like to thank my Labmates at SeaLab who provided daily smiles at work and great interactions and feedback about research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Regular expressions (*regexes*) are an important software engineering tool. They are used in an estimated 30–40% of open-source projects [20, 29]. Regexes use spans the system stack, including for security-specific tasks (*e.g.,* input validation [15, 85], web application firewalls [2], and anti-virus [63]); and for general-purpose tasks (*e.g.,* text processing [42, 55] and syntax highlighting [22, 36]). Regexes may introduce a security vulnerability: *Regular Expression Denial of Service (ReDoS)* [27, 68]. ReDoS is an algorithmic complexity vulnerability [28], denying service by exhausting computational resources. In a ReDoS exploit, a regex's super-linear worst-case execution time is triggered during regex matching. ReDoS can affect software that uses an unsafe (*backtracking*) regex engine to evaluate a problematic (*super-linear*) regex on untrusted strings (such as user input). Unsafe regex engines are common in practice, including the regex engines shipped in many popular programming languages, *e.g.,* Java, JavaScript, Ruby, and Python [25, 31]. This hazardous infrastructure exposes many web services to ReDoS. For example, ReDoS caused service outages at Stack Overflow in 2016 [37] and at Cloudflare in 2019 [46]. Beyond those cases, Davis *et al.* showed that the software supply chain has many vulnerable regexes [29], and Staicu & Pradel found that those regexes render hundreds of popular websites vulnerable to ReDoS [73].

Given the severity and prevalence of ReDoS, researchers have proposed mechanisms to defend against it. The majority of them apply to the software maintenance stage —to automatically identify [16, 52, 56, 65, 70, 77, 78, 88, 89, 90] and fix [23, 24, 54, 82] vulnerable regexes— or

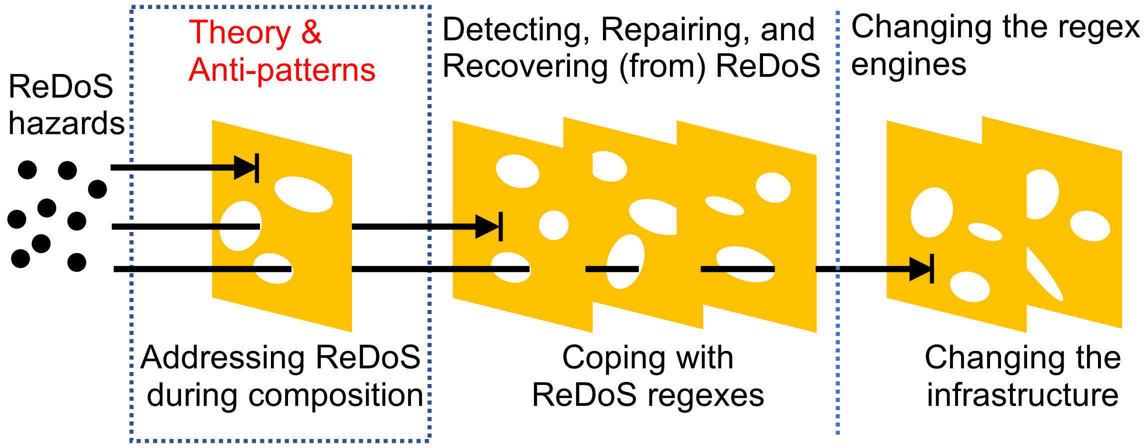


Figure 1.1: The Swiss cheese model [66, 67] (defense in depth) for ReDoS. Most existing techniques to address ReDoS-vulnerable regexes are applied during their maintenance or after their deployment. Our work provides early prevention, avoiding the composition of vulnerable regexes in the first place.

after software deployment [14, 26, 33].

In this paper, we set a different goal: **preventing the introduction of ReDoS in the first place** —at regex composition time, as illustrated in Figure 1.1. We propose a novel theory of regex infinite ambiguity (IA) —a necessary component of ReDoS— and derive a collection of anti-patterns from it. We thus leverage our foundational contribution to enable a practical one. Developers can apply our anti-patterns at composition time to avoid introducing infinitely ambiguous (IA) regexes in their source code, and thus avoid ReDoS.

Our proposed theory and anti-patterns fill a gap in the existing multi-layer defense against ReDoS, providing a coding standard to prevent it at composition time instead of at maintenance or deployment time. Our approach aligns with the multi-layer approach to software security known as *defense in depth* [75], or the *Swiss cheese* approach [51, 66, 67]. When using many layers to defend against security problems, the strengths of each layer can counterbalance the limitations of the others. Also, different defense layers may better fit the workflow of different software development teams.

To the extent of our knowledge, only two existing approaches may (partially) support developers to avoid composing IA regexes, both of which are limited. First, Brabrand & Thomsen's theory [18] characterizes ambiguous regexes, but does not distinguish between finite (safe) and infinite (unsafe) ambiguity. Our proposed theory improves upon Brabrand & Thomsen's by soundly and completely characterizing infinitely ambiguous regexes. Second, existing anti-patterns in reference texts characterize regexes vulnerable to ReDoS [39, 44, 45, 76]. However, a recent study found that they are insufficient —they have a large ratio of false positives [29]. Our proposed anti-patterns improve upon existing anti-patterns by being derived from a formal theory of infinite ambiguity, and achieving high precision and recall as a result.

As a prevention measure, our proposed work also follows the growing trend in software security of *shifting left* security concerns —addressing them early in the life-cycle to achieve benefits in cost savings, quality control, and speed to market *e.g.,* [4, 5, 6]. More generally, in software engineering, it is common wisdom that problems detected earlier are easier and cheaper to fix —driving common practices like continuous integration [47] and modern code review [12]. Also, since we designed our anti-patterns for developer comprehension, we contribute to the important mission of educating software developers in secure coding —as a valuable complement to automated security tools [41].

We evaluated our proposed approaches. For our IA regex theory (chapter 5), we proved (Appendix A) its soundness and completeness. To evaluate our anti-patterns (chapter 6), we performed two complementary experiments (chapter 7 and chapter 8). First, we evaluated the effectiveness of our anti-patterns for comprehensiveness (chapter 7) by applying them to a large-scale dataset of real-world regexes [31] Our experiments showed that our anti-patterns provide higher precision and recall than existing anti-patterns —100% vs. 50% precision, and 99% vs. 87% recall. We also learned that two of our six anti-patterns (Concat 1 and

Concat 2) are enough to cover a wide majority of IA regexes (99%) in practice. Second, we also evaluated the usability of our anti-patterns at regex composition time (chapter 8). Our human-subjects experiment ($N = 20$) showed that all our studied practitioners (100%) successfully applied our anti-patterns during regex composition for all our 5 studied tasks, greatly improving the success that they achieved when applying existing anti-patterns (50%).

This paper provides the following contributions:

1. A novel theory that soundly and completely characterizes regex infinite ambiguity (chapter 5).

2. A collection of (IA) anti-patterns of regex infinite ambiguity, derived from our theory, to apply at composition time to avoid inserting regexes vulnerable to ReDoS (chapter 6).

3. An evaluation of the comprehensiveness of our proposed IA anti-patterns over a large-scale of real-world regexes, showing that they could capture the wide majority of IA regexes (chapter 7).

4. A human-subjects evaluation, showing that our IA anti-patterns are also effective when applied in practice by developers to compose regexes (chapter 8).

Chapter 2

Background

This section provides background on regexes (§2.1) and ReDoS (§2.2), describes the impact of ReDoS in practice (§2.3), and discusses the threat model (§2.4).

2.1 Regular Expressions (Regexes)

2.1.1 Regexes

A regex is a compact descriptive format used to specify a language: *i.e.,* a set of strings. Given a finite alphabet of terminal symbols, Σ, and the metacharacters, '$|$', '$\cdot$', and '$*$', the regex syntax is [71]:

$$R \to \phi \;\Big|\; \epsilon \;\Big|\; \sigma \;\Big|\; R_1|R_2 \;\Big|\; R_1{\cdot}R_2 \;\Big|\; R_1*$$

where ϕ denotes the empty language; ϵ is the empty string; the characters $\sigma \in \Sigma$ are terminal symbols; $R_1|R_2$ is alternation (union); $R_1{\cdot}R_2$ is concatenation; and $R*$ is zero or more repetition (Kleene closure [53]). The language function $L : R \to 2^{\Sigma^*}$ gives semantics:

$$\begin{array}{ll} L(\phi) = \phi & L(R_1|R_2) = L(R_1) \cup L(R_2) \\ L(\epsilon) = \{\epsilon\} & L(R_1{\cdot}R_2) = L(R_1){\cdot}L(R_2) \\ L(\sigma) = \{\sigma\} & L(R*) = L(R)* \end{array}$$

We refer to the above "regular" regexes, including alternation, concatenation, repetition, and "syntactic sugar" extensions, as (Kleene's) K-regexes. These "syntactic sugar" extensions include character ranges `[a-c]` (equivalent to `a|b|c`); optional regex `R?` (as `R|`ϵ); and various forms of repetition such as `R+` (equivalent to `RR*`) and `R{2,3}` (equivalent to `RR|RRR`).

Outside the scope of K-regexes, there are also non-regular features. The most commonly supported non-regular features are lookaround assertions, backreferences, atomic groups, and possessive quantifiers [39]. These are semantically complex and their use is rare [20, 32]. Our theory and anti-patterns therefore support K-regexes but currently exclude non-regular features.

2.1.2 Regex Ambiguity

The regex language semantics allow membership to be checked using a parser. A regex is *ambiguous* if there is a string in its language that can be matched with more than one parse tree [18, 78]. For example, the regex `a|a` can parse the input "a" in two ways, *i.e.,* yielding two parse trees, one using the left `a` and one the right.

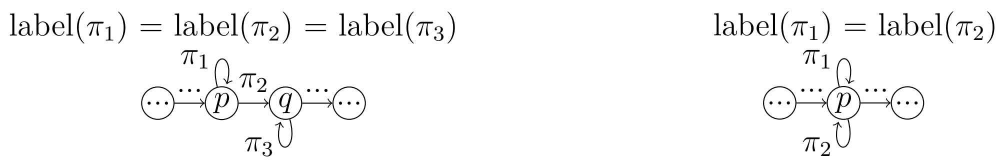
For K-regexes, a regex match can be equivalently performed by simulating an input on a non-deterministic finite automaton (NFA) constructed from the regex (*e.g.,* using Thompson's algorithm [80]). To simplify discussion, we will reason about regex ambiguity over an equivalent, ambiguity-preserving, ϵ-free NFA [86, 89] (with no ϵ-loops). From the NFA perspective, a regex's NFA is ambiguous if there is a string that can be accepted along multiple paths of the NFA.

2.1.3 Infinitely Ambiguous (IA) Regex

Regexes can have various degrees of ambiguity: no ambiguity (deterministic); finitely ambiguous (bounded ambiguity regardless of input length); or infinitely ambiguous (polynomial or exponential) in the length of the input [10, 74]. A regex is infinitely ambiguous if it has a sub-regex (equivalently, an NFA section) with the *infinite-degree-of-ambiguity (IDA)* property [86]. Given an ϵ-free finite automaton A, the necessary and sufficient conditions for A to be infinitely ambiguous are given by Weber & Seidl [86]. Next we summarize these conditions.

Figure 2.1(a) illustrates how a *polynomially IDA (PDA)* section appears in the NFA of a polynomially ambiguous regex. A substring $label(\pi_i)$ can be matched in the loop π_1 at node p, the path π_2 from p to q, or another loop π_3 at q. For example, consider the regex `a*a*` for an input "$aa...a$" of length N. As any two partitions of the input can be matched with the first `a*` and second `a*`, there are N matching paths. Each of those paths costs $\mathcal{O}(N)$, so the total time complexity is quadratic in the size of input $\mathcal{O}(N^2)$.

Figure 2.1(b) illustrates an *exponentially IDA (EDA)* section in the NFA of an exponentially ambiguous regex. A substring $label(\pi_i)$ can be matched in either of two loops π_1 or π_2 at node p. The regex `(a|a)*` is a representative example of exponentially ambiguous regexes. Each 'a' of the input "$a...a$" can be matched by either the upper or lower loop, and thus the total number of matching paths becomes 2^N, leading to an exponential time complexity $\mathcal{O}(2^N)$.


label(π_1) = label(π_2) = label(π_3)
π_1
π_2
p
q
π_3
label(π_1) = label(π_2)
π_1
p
π_2

(a) PDA (or IDA_d) (b) EDA

Figure 2.1: Illustration of Polynomial and Exponential Degree of Ambiguity (PDA, EDA) in the NFA [86].

2.2 Regex-Based Denial of Service (ReDoS)

Regex-based Denial of Service (ReDoS) [27, 33] is a security vulnerability by which a web service's computational resources are diverted from legitimate client interactions into an expensive regex match, degrading its quality of service. Following the presentation of Davis *et al.* [33], ReDoS is caused by a combination of three Conditions:

(C1) a *backtracking regex engine* used in evaluation, and

(C2) a *super-linear (SL) regex*, applied to evaluate

(C3) a *malign input*.

2.2.1 Backtracking Regex Engine

Most regex engines (*e.g.,* versions of PHP, Perl, JavaScript, Java, Python, Ruby, and C#) use a backtracking search algorithm, *e.g.,* Spencer's [72], to determine whether an input is in the language described by a regex [25, 31]. The typical implementation has good common-case performance but can be problematically slow in the worst case — *e.g.,* when it evaluates a *super-linear (SL) regex* with *malign input* [27, 28, 29].

2.2.2 Super-linear (SL) Regex

An SL regex is an IA regex whose NFA has a *prefix* section, followed by an IDA section (either PDA or EDA), followed by a *suffix* section [90]. The IDA section is the root cause of the regex's super-linear behavior. The prefix must be considered to reach this IDA section, and the suffix must typically lead to a mismatch in order to trigger backtracking.

2.2.3 Malign Input

A malign input triggers the super-linear behavior of an SL regex by driving the backtracking engine into evaluating a polynomially or exponentially large number of possible NFA paths, exhausting available resources. Following the structure of an SL regex, a malign input is composed of three sections: a *prefix*, then a *pump*, then a *suffix*. The *prefix* brings the engine to the IDA section of the regex, the pump produces polynomially or exponentially many paths to explore, and the *suffix* causes the engine to mismatch. This drives the engine to backtrack through the many paths [16].

2.2.4 SL vs. IA Regexes

Infinite ambiguity is a necessary but insufficient condition for super-linear behavior. For example, the regex `(a*)*.*` is IA but not SL. Its final sub-regex `.*` accepts any string, so no mismatch-triggering suffix exists.

2.3 The Impact of ReDoS in Practice

The impact of a slow regex varies depending on how it is used in a web service. For example, the company Cloudflare used a regex as part of deep packet inspection. When it deployed a quartic regex in 2019, its network routing servers spent all of their time on a regex match, effectively halting their CDN service [46]. As another example, the popular Q&A website Stack Overflow used a quadratic regex for modest server-side rendering. When a long problematic post reached the homepage, access to the homepage took several seconds, causing their monitoring system to mark each of their backend servers inoperable in succession. This reduced their load-balancing capacity, eventually leading to an outage [37].

Beyond these well-publicized examples of ReDoS, we examined CVE data for another perspective on the impact of ReDoS in practice. Figure 2.2 shows the increasing trend of ReDoS CVEs each year since 2010.

2.4 Threat Model

We suppose the following threat model for ReDoS, aligned with the common use of regexes for input sanitization in web software [20, 60, 90]. The victim's regex engine uses a backtracking regex engine (ReDoS Condition 1), which is common for many server-side programming languages. The victim uses a regex to sanitize the untrusted input (Condition 2). The attacker controls the input string (Condition 3). The input string could be arbitrary long. Bounding its length from the server side is not effective [29]. Both Stack Overflow and Cloudflare cases processed user-controlled strings containing kilobytes of text.

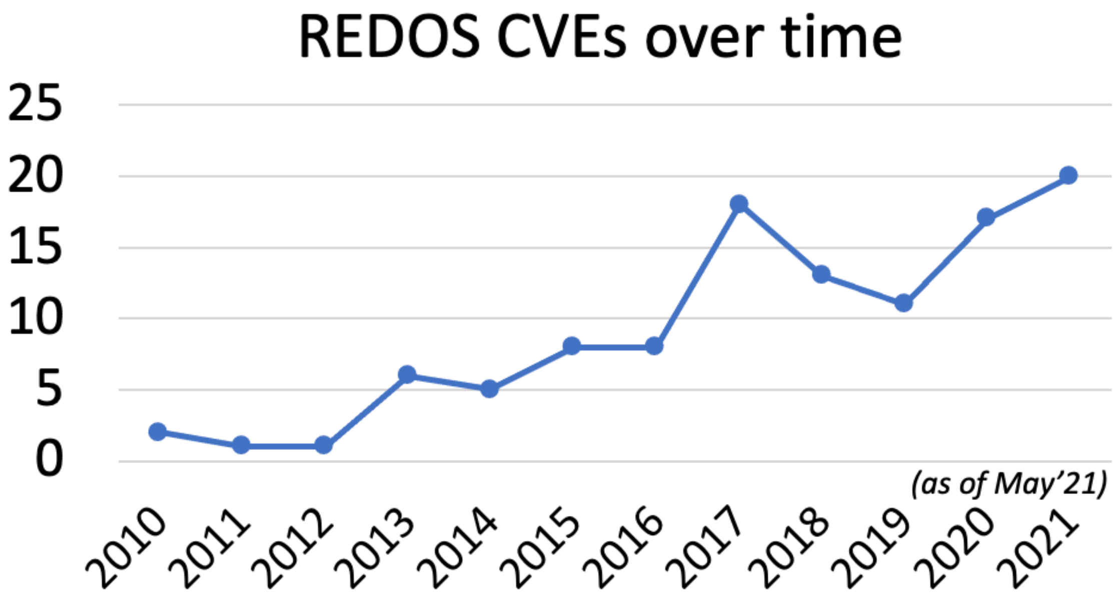


Figure 2.2: CVEs due to ReDoS since 2010. The data were obtained by a two-step process: a preliminary labeling of the CVE database using key words and phrases (*e.g.*, "ReDoS" or "extremely long time" with a reference to regular expressions), followed by a manual inspection for accuracy.

Chapter 3

Review of Literature

Regexes are a ubiquitous tool used in security-sensitive contexts. Given the importance of regexes, researchers have examined the ReDoS problem from many perspectives.

3.1 Empirical measurements of ReDoS in practice

Although the ReDoS attack was proposed twenty years ago by Crosby and Wallach [27, 28], researchers have only recently attempted to estimate its impact. In 2018, Davis *et al.* reported that SL regexes were present in many popular open-source software modules, and that engineers struggled to repair them [29]; in 2019 they observed that these regexes displayed super-linear (SL) behavior in the built-in regex engines used in most mainstream programming languages [31, 32]. Concurrently, Staicu & Pradel showed that 10% of Node.js-based web services were vulnerable to ReDoS due to their use of vulnerable npm modules [73].

These findings motivated substantial further research into the ReDoS problem. Web services must process user input (ReDoS condition 3), so researchers have examined infrastructure-level solutions (ReDoS condition 1) and application-level solutions (ReDoS condition 2).

3.2 Changing the Infrastructure

3.2.1 Changing the regex engine

There are both classic and more recent alternatives to the exponential-time backtracking regex algorithm. The earliest published regex matching algorithms, by Brzozowski in 1964 [19] and Thompson in 1968 [80], offer linear-time guarantees. There are production-grade implementations of Thompson's approach, notably Google's RE2 regex engine [26] and the engines in Rust [35] and Go [43]. Microsoft has considered Brzozowski's approach for .NET [69], as well as deterministic matching strategies [48, 81]. However, these algorithms are difficult to adopt in legacy programming languages, since it does not provide support for non-regular regex features. Davis *et al.* presented an alternative approach that preserves the backtracking framework [72], but reduces the time complexity to linear via memoization [33]. While this approach is sound, it trades space for time and may thus simply exhaust another resource instead.

3.2.2 Abandoning regexes

To avoid ReDoS, some advocate alternative approaches to string matching. Davis *et al.* reported that reverting to built-in string functions was a popular ReDoS repair technique [29]. However, regexes exist for a reason — they concisely express long sequences of string operations. To safely benefit from expressiveness, the most prominent regex alternative is parsing expression grammars (PEGs) [38]. The Rosie Pattern Language simplifies converting regexes into PEGs [49]. Although this approach is attractive, technical inertia suggests that regexes are here to stay.

3.3 Coping With SL Regexes

The majority of ReDoS research assumes that SL regexes are already present in a web service, and considers various coping mechanisms.

3.3.1 Detecting SL Regexes

Many approaches have been proposed to automatically identify super-linear regexes. Many researchers statically analyze the NFA of the regex. Berglund *et al.* [16] defined a prioritized type of NFA to simulate a backtracking engine in Java and decide if a regex will show SL behavior. Weideman *et al.* [88, 89] also use a prioritized NFA and look for IDA in it. Wustholz *et al.* [90], also look for the IDA pattern in the NFA and compute an attack automaton that produces attack input strings. Liu *et al.* [56] adds support for modeling and analyzing less common regex features, *e.g.,* set operations.

Others statically analyze different representations of the regex. Kirrage *et al.* [52] analyze an abstract evaluation tree of the regex. Rathnayake *et al.* [65] look for exponential branching in the regex evaluation tree. Sugiyama *et al.* [77] analyzes the size of a tree transducer for the regex. Finally, Sulzmann *et al.* [78] use Brzozowski derivatives to create a finite state transducer to generate parse trees and minimal counter-examples.

Finally, other approaches apply dynamic analysis to detect SL regexes. Shen *et al.* [70] and McLaughlin *et al.* [58] proposed search algorithms for inputs with super-linear matching time. The broader category of algorithmic complexity detectors, *e.g.,* [17, 59, 62, 64, 87], can also be extended to detect ReDoS.

3.3.2 Repairing SL Regexes

Other approaches focus on repairing SL regexes to make then non-SL. These approaches offer trade-offs for the repaired regex, in: semantic similarity, (perceived) readability, and support for uncommon features. Van der Merwe *et al.* presented a modified flow algorithm to convert an ambiguous K-regex into an equivalent unambiguous one [82], with perfect semantic equivalence, but lower readability. Cody-Kenny *et al.* proposed evolving a regex towards a higher-performing alternative — though evaluated on average-case performance, this scheme may also work for worst-case performance [24], supporting uncommon features but with only approximate semantics. More recently, Li *et al.* [54] proposed an approach to repair SL regexes with deterministic regex constraints to avoid regex ambiguity. This approach uses a human-in-the-loop to provide good examples. Finally, Claver *et al.* [23] proposed a synthesis-based approach for repairing SL regexes.

3.3.3 Recovering From ReDoS

After a system containing a ReDoS vulnerability is deployed, it is possible to detect and isolate ReDoS attacks to minimize the damage incurred. Bai *et al.* proposed the only ReDoS-specific approach of this kind, applying deep learning to detect attack strings and redirect them to a sandbox [14]. More general approaches that consider anomalous resource utilization, *e.g.,* time [30], CPU [59], or other resources [34].

3.4 Addressing ReDoS during Regex Composition

While substantial study has been made of dealing with SL regexes *after* they enter a code-base, only two existing approaches (partially) support developers at composition time. The philosophy of these approaches is to avoid composing IA regexes in the first place. Davis *et al.* characterized a set of practitioner anti-patterns to avoid ReDoS, but reported that these have a high false positive rate [29]. Alternatively, Brabrand & Thomsen developed theories to identify *un*ambiguous regexes, and treat all other regexes as suspect, including both SL regexes and merely finitely ambiguous (*i.e.,* non-vulnerable) regexes [18].

Our proposed work provides important improvements over these two approaches. In contrast with Brabrand & Thomsen's, our theory distinguishes between finite and infinite ambiguity, enabling developers to distinguish between likely-unproblematic (non-IA) and problematic (IA) regexes. In contrast with practitioner-prescribed anti-patterns, we provide a theoretical grounding to formally capture their limitations, and we provide new anti-patterns that provide higher precision and recall capturing IA regexes.

Chapter 4

Problem Statement and Proposed Approach

4.1 Problem Statement

In light of the gap in the first stage of the Swiss cheese model (Figure 1.1), our goal is to support developers to avoid composing regexes vulnerable to ReDoS. An ideal solution must accomplish this while honoring three additional properties drawn from regex engineering practices.

Lightweight: First, developers may compose and evolve multiple drafts before settling on a regex to commit into their code base [13, 60]. They also often consider multiple regex candidates to reuse, from multiple sources (*e.g.,* StackOverflow, or other software projects) [60], and in multiple programming languages [31]. Thus, developers wanting to avoid composing SL regexes need a technique that they can apply in an agile way to assess multiple (alternative or refined) candidate regexes coming from diverse programming languages and sources, some of which may be incomplete, incorrect, or not in an executable environment (*e.g.,* in StackOverflow).

Regex-modality-based: Second, even if there are many ways to describe regex behavior mathematically, including transducers, NFAs, and Brzozowski derivatives, many developers can only understand the regex modality [13]. Therefore, any approach to help developers avoid composing SL regexes and aiming to support a wide population of developers should analyze and explain its results in the regex modality.

Conservative: Finally, developers evolve regexes over time in their codebases, often by extending their language [84]. Therefore, a technique supporting developers to avoid composing SL regexes is ideally conservative in the regexes that it identifies as risky —potentially also including regexes that may not be currently SL, but are likely to become SL after being modified.

4.2 Proposed Approach

Our work aims to help developers to avoid introducing SL regexes. For that goal, we propose a novel theory of regex infinite ambiguity (IA) (chapter 5), and we derive a collection of anti-patterns from it (chapter 6).

Lightweight: To support developers in diverse scenarios of regex composition, we propose a theory and anti-patterns that developers can learn and apply manually in their head, as they assess draft regexes (complete or not, correct or not, executable or not, and in any language).

Regex-modality-based: To support a wide population of developers, our theory and anti-patterns let developers assess regexes in the regex modality. For many developers, it is the only one that they understand [84].

Conservative: To provide a conservative assessment of the risk of regexes, our theory and anti-patterns help developers identify IA regexes (instead of SL ones). IA regexes capture all SL regexes —they are a necessary condition for them. Also, non-SL IA regexes can accidentally become SL during maintenance [84], *e.g.,* by potentially introducing a mismatch-triggering suffix that makes them SL. As a final benefit, detecting IA regexes is likely simpler for developers to apply, since it requires fewer conditions.

4.3 The Nature of our Contributions

Our theory is a foundational contribution: it provides a set of rules to soundly and completely describe the characteristics of IA regexes. Our anti-patterns are a practical contribution: a simplified articulation of our theory into a collection of anti-patterns that developers can apply to assess whether their regex under composition is IA. We discuss how these approaches complement existing ReDoS defenses in chapter 9.

4.4 Evaluation Design

Next, we prove our regex IA theory (chapter 5 and Appendix A). We then evaluate our regex IA anti-patterns in two complementary ways: in comprehensiveness (chapter 7) and in usability (chapter 8). For comprehensiveness, we measured their effectiveness to capture IA over a diverse collection of real-world regexes. For usability, we studied how they helped developers wen composing regexes, in a human subjects experiment.

Chapter 5

Theory of Regex Infinite Ambiguity

This section introduces an existing theory of regex ambiguity [18], discusses its limitations, and presents our novel theorems.

Recalling chapter 2, a regex with an infinite degree of ambiguity (IA) [86] has the necessary condition for super-linear (SL) regex behavior [27, 28, 29]. Though the NFA-level conditions for IA regexes (namely PDA and EDA regions) are well known [86], we lack characterizations in terms of regex syntax and semantics. We provide such a description to help developers avoid ReDoS at regex composition time.

5.1 Preliminaries

Brabrand & Thomsen [18] developed the state of the art description of regex-level ambiguity. They introduced an overlap operator, $\bowtie$, between two languages $L(R_1)$ and $L(R_2)$. The set $L(R1) \bowtie L(R2)$ contains the ambiguity-inducing strings that can be parsed in multiple ways across $L(R1)$ and $L(R2)$. More formally, with $X = L(R_1)$ and $Y = L(R_2)$,

$$X \bowtie Y = \{xay \mid x, y \in \Sigma^* \land a \in \Sigma^+ \land x, xa \in X \land ay, y \in Y\}$$

Using this operator, Theorem 0 summarizes their findings.

Theorem 0 (Brabrand & Thomsen [18])**.** Given unambiguous regexes R_1 and R_2:

(a) $R_1|R_2$ is unambiguous iff $L(R_1) \cap L(R_2) = \phi$.

(b) $R_1{\cdot}R_2$ is unambiguous iff $L(R_1) \bowtie L(R_2) = \phi$.

(c) R_1* is unambiguous iff $\epsilon \notin L(R_1) \wedge L(R_1) \bowtie L(R_1*) = \phi$.

In their implementation of this Theorem, Brabrand & Thomsen use Møller's BRICS library [61], and actually rely on what we call the Møller overlap operator, Ω. We use this operator in our theorems. The Møller overlap operator describes only the ambiguous core "a":

$$X\ \Omega\ Y = \{a \mid a \in \Sigma^+ \wedge \exists\ x, y \in \Sigma^* s.t.\ x, xa \in X \wedge ay, y \in Y\}$$

Limitation: Given unambiguous regex components, Theorem 0 specifies when a composed regex remains unambiguous. Yet not all ambiguity is harmful. For example, the regex `\w|\d` is finitely ambiguous. This may improve readability [9]; it is not a ReDoS risk.

5.2 Regex Infinite Ambiguity Theorems

This section presents our regex ambiguity theory for composition with alternation (Theorem 1), concatenation (Theorem 2), and star (Theorem 3). We give proof sketches, examples, and the ReDoS implications. Full proofs are in Appendix A.

Theorem 1 (Ambiguity of Alternation)**.** Given unambiguous regexes R_1 and R_2,

(a) $R_1|R_2$ is finitely ambiguous iff $L(R_1) \cap L(R_2) \neq \phi$.

(b) $R_1|R_2$ cannot be infinitely ambiguous.

Proof Sketch: The theorem states that given unambiguous regexes R_1 and R_2, if $R_1|R_2$ is ambiguous, then it is always finitely ambiguous. Since R_1 and R_2 are both unambiguous, for any matching input w, there is only one path through R_1 and R_2. Therefore, for $R_1|R_2$ and any matching input w, there are at most two matching paths.

Example: For the regex `a*|a*`, consider an input "*a...a*" of length N. Regardless of the input length, the number of accepting paths will be at most 2: *i.e.,* the first $a*$ or the second $a*$.

ReDoS Implications: If two regexes R_1 and R_2 are unambiguous, $R_1|R_2$ is always safe (does not form IA).

Theorem 2 (Ambiguity of Concatenation)**.** Suppose unambiguous regexes R_1 and R_2, and that $L(R_1) \between L(R_2) \neq \phi$ (so $R_1 \cdot R_2$ is ambiguous by Theorem 0). Then:

(a) $R_1 \cdot R_2$ is infinitely ambiguous iff $L(R1)$ contains the language of a regex $BC*D$ and $L(R2)$ contains the language of a regex $EF*G$, where $\epsilon \notin L(C) \wedge \epsilon \notin L(F) \wedge L(C) \cap L(F) \cap L(DE) \neq \phi$.

(b) Otherwise, $R_1 \cdot R_2$ must be finitely ambiguous.

Proof Sketch: $\Longleftarrow$: Consider the string "*bcc...cdeff...fg*" $\in L(R1 \cdot R2)$ which can be divided into two strings "*bcc...cd*" $\in L(BC*D) \subset L(R1)$ and "*eff...fg*" $\in L(EF*G) \subset L(R2)$ where

$c = f = de$. By hypothesis, the string "*dede...de*" can be matched arbitrarily many times in R_1 (by `C*`) and in R_2 (by `F*`). We can choose an arbitrarily long string and obtain arbitrary ambiguity in $R_1 \cdot R_2$.

$\implies$: Suppose $R_1 \cdot R_2$ is infinitely ambiguous. The NFA corresponding to $R_1 \cdot R_2$ cannot contain the EDA structure because this requires a self-loop — *i.e.,* that R_1 or R_2 is already ambiguous. Therefore the NFA of $R_1 \cdot R_2$ must contain an PDA structure, as shown in Figure 2.1(a). We can map the two loops π_1 and π_3 with `C*` and `F*` respectively; and the bridge π_2 with $D \cdot E$ in the regex representation, where $L(C) \cap L(F) \cap L(DE) \neq \phi$.

Example: For `(a*a)(aa*)` and the input "*aa...a*" of length N, it has N accepting computations, one for each of the indices of the input at which to split the string into a left half consumed by R_1 and a right half consumed by R_2.

ReDoS Implications: Though two regexes R_1 and R_2 are unambiguous, $R_1 \cdot R_2$ could be IA, thus concatenation should be used with care. Theorem 2(a) implies that for $R_1 \cdot R_2$ to be IA, there must be a star component in both R_1 and R_2. In §6.1, we introduce three forms of concatenation anti-patterns based on this observation.

Theorem 3 (Ambiguity of Star)**.** Given unambiguous regex R,

(a) $R*$ is infinitely ambiguous iff $\epsilon \in L(R) \vee L(R)\ \Omega\ L(R*) \neq \phi$.

(b) $R*$ cannot be finitely ambiguous.

Proof Sketch: The theorem states that given an unambiguous regex R, if $R*$ is ambiguous, then it is always infinitely ambiguous. Suppose $R*$ is ambiguous. Then there is some input w that it can match in k ways, $k > 1$. So there is an input ww that it can match in $k * k = k^2$ ways. The degree of ambiguity increases as a function of input length: *i.e., R∗* is infinitely ambiguous.

Example: For the regex `(a*)*`, consider an input "*aaa...a*" of length N. There are two ways (inner `*` or outer `*`) to match each '*a*' of the input, making the total number of ways to match to be 2^N.

ReDoS Implications: Even though an original regex R is unambiguous, $R*$ can be IA. Later in §6.2, we introduce an anti-pattern that only checks for a subset of conditions for simplicity.

Theorem 4. Given a finitely ambiguous regex R, $R*$ is always infinitely ambiguous.

Proof Sketch: The proof follows the same logic as Theorem 3.

Example: For the regex `(a|a)*`, consider an input "*aaa...a*" of length N. There are two ways (first `a` or second `a`) to match each '*a*' of the input, making the total number of ways to match to be 2^N.

ReDoS Implications: If R is finitely ambiguous, from alternation ($R = P|Q$) or concatenation ($R = P{\cdot}Q$), $R*$ is always IA. Later in §6.2, we introduce two anti-patterns of the form $(P|Q)*$.

Chapter 6

Anti-patterns for Regex Infinite Ambiguity

This chapter describes the proposed anti-patterns for IA regexes (IA anti-patterns, in short), derived from the above theory of regex infinite ambiguity. Ideally, anti-patterns should be sound and complete as the theory, yet usable and simple as existing anti-patterns. With the two goals in mind, we extracted six IA anti-patterns from the theory that we thought are common and easy to describe. They were iteratively refined through internal discussion between several of the authors. Table 6.1 summarizes our proposed IA anti-patterns and examples. Since alternation itself cannot make a regex IA (Theorem 1), there is one group of concatenation anti-patterns derived from Theorem 2 and another group star anti-patterns from Theorem 3. Later in chapter 7 and chapter 8, we quantitatively and qualitatively compare our anti-patterns with state-of-practice (SOP) anti-patterns.

6.1 Concatenation Anti-patterns

The Concat anti-patterns are based on Theorem 2, stating that a concatenated regex $R = R_1 \cdot R_2$ becomes IA if $L(R1)$ contains the language of a regex $BC * D$ and $L(R2)$ contains the language of a regex $EF * G$, where $L(C) \cap L(F) \cap L(DE) \neq \phi$. In other words, $L((BC * D)(EF * G)) \subset L(R)$. This implies that R has a sub-regex of the form $P * SQ*$ in which $P*$

Table 6.1: Description of our proposed IA anti-patterns with examples.

Anti-pattern	**Description**	**Example**
Concat 1	`R = ...P*Q*...` (`R` has a sub-regex `P*Q*`): The two quantified parts `P*` and `Q*` can generate some shared string *s*.	`a*(aa)*` : both can generate *aa*
Concat 2	`R = ...P*SQ*...` (`R` has a sub-regex `P*SQ*`): The two quantified parts `P*` and `Q*` can generate a string *s* from the middle part *S*.	`(a|b)*ab(ab)*` : quantified parts `(a|b)*` and `(ab)*` can generate the middle part *ab*.
Concat 3	`R = ...P*S*Q*...` (`R` has a sub-regex `P*S*Q*`): Advanced form of Concat 1. Since `S*` includes an empty string, the ambiguity between `P*` and `Q*` can be realized.	`a*b*a*`, `b*` can be skipped.
Star 1	`R*`, `R= (P|Q|...)`: There is an intersection between any of the two alternates i.e., both can generate some shared strings.	`(\w|\d)*`, both can generate digits [0-9].
Star 2	`R*`, `R= (P|Q|...)`: You can make one option of the alternation by repeating another option multiple times or by concatenating two or more options multiple times.	`(x|y|xy)*`, we can create the 3rd option *xy* by adding the first option *x* and second option *y*.
Star 3	`R*`, `R= (...P*...)`: Nested quantifiers, but only if `RR` would follow any of the concat antipatterns above.	For `(a*)*`, we write a*a*, it is infinitely ambiguous by Concat 1. For `(xy*)*`, we write `xy*xy*`, it is not infinitely ambiguous by any of the concat anti-patterns.

and $Q*$ are mapped to the $C*$ component and $F*$ component respectively, and S is mapped to the bridge component DE between $C*$ and $F*$.

Concat-1: The anti-pattern where $P * Q*$ is a sub-regex of R represents a simplest form without the bridge S. Developers only need to consider if there is a string matched in both $P*$ and $Q*$.

Concat-2: The anti-pattern where $P * SQ*$ is a sub-regex of R has the bridge S component. Developers need to find a string matched in all $P*$, $Q*$, and S.

Concat-3: The anti-pattern where $P * S * Q*$ is a sub-regex of R represents the case with the optional bridge S. As in Concat-1, developers consider for a string matched in both $P*$ and $Q*$.

Gap Analysis: The Concat anti-patterns represent three different ways that the bridge component DE (from Theorem 2) may appear as a sub-regex of the form ϵ, S, or $S*$. Thus, there is no gap between theory and anti-patterns.

6.2 Star Anti-patterns

The Star anti-patterns stem from Theorem 3 and Theorem 4.

Star-1 and Star-2: These anti-patterns are designed to prevent (some) regexes of the form $R*$ where $R = (P|Q|...)$. Theorem 4 states that if R is finitely ambiguous, then $R*$ becomes IA. From Theorem 1(a), alternations may introduce finite ambiguity. The Star-1 and Star-2 anti-patterns describe two common conditions when the subregex $(P|Q|...)$ becomes finitely ambiguous.

Gap Analysis: There is a gap between Theorem 4 and the Star-1 and Star-2 anti-patterns. Our anti-patterns do not consider all possible forms of finitely ambiguous regexes. For instance, the concatenation may also introduce finite ambiguity (Theorem 2(b)). Thus, some regexes of the form $(P{\cdot}Q)*$ could be IA as well: *e.g.,* `((a|ab)(c|bc))*`. Regexes under these missing conditions would appear as false negatives for these anti-patterns.

Star-3: This anti-pattern prevent (some) regexes of the form $R*$ where R has a sub-regex of the form $P*$. Theorem 3 states the conditions when $R*$ becomes IA. considering the first condition $\epsilon \in L(R)$ is relatively trivial. Yet, the second condition $L(R)\Omega L(R*) \neq \phi$ requires reasoning about a language overlap between $L(R)$ and an arbitrary repetition of $L(R*)$, which could be tricky. Based on the common knowledge that a nested quantifier (*e.g.,*

$(P*)*)$ is bad [76], the Star-3 anti-pattern only considers the case where R has a sub-regex $P*$, a more general form of nested quantifiers. The Star-3 anti-pattern further simplifies the condition and asks developers to consider the overlap between $L(R)$ and (twice-repeated) $L(R{\cdot}R)$, using the above Concat anti-patterns.

Gap Analysis: The Star-3 anti-pattern does not incorporate all the conditions in Theorem 3. Thus, regexes under the missing conditions would appear as false negatives for this anti-pattern.

Chapter 7

Experiment 1: Comprehensiveness

Our proposed IA anti-patterns (chapter 6) are backed by our theory (chapter 5 and Appendix A). Next, we further validated our IA anti-patterns with an automated experiment. We evaluated how effective our proposed (IA) anti-patterns would be in identifying IA regexes among a broad set of regexes. —over a large-scale dataset of 209,188 real-world regexes [31]. To validate our IA anti-patterns over such a large set of regexes, we automated their application —by implementing an experimental prototype in the form of automatic scripts. To provide a point of reference, we also implemented scripts for the existing state-of-the-practice (SOP) anti-patterns, and we also executed them over the same dataset. We then measured the effectiveness of both sets of anti-patterns using precision and recall.

7.1 Studied Techniques

We studied two sets of anti-patterns in this experiment:

7.1.1 Our Proposed (IA) Anti-patterns

For each anti-pattern, we implemented a script in Java that checks if a given regex is IA according to it. We parsed regexes in the PCRE format, using the ANTLR 4 grammar and parser for PCRE [1]. We used the BRICS [61] tool to measure whether multiple sub-regex parts can generate any shared string.

We implemented our IA anti-patterns to support K-regexes, *i.e.,* all regex features except for non-regular ones, since they are complex and rare (see §2.1.1). We also excluded extended character classes such as POSIX and Unicode (for simplicity). These non-regular features and extended character classes are also not supported by most past approaches to automatically identify ReDoS during software maintenance [16, 52, 65, 77, 78, 88, 89, 90].

Our implementation supports 450,753 regexes (83.8%) of our studied large-scale dataset (all but 10% with non-regular features, 6% not supported by BRICS, and 0.2% with extended character classes). This level of completeness is comparable to prior research prototypes for regex analysis [33, 65, 90].

7.1.2 State-of-the-practice (SOP) Anti-patterns

To provide a point of reference, we also executed a set of state-of-the-practice (SOP) anti-patterns that are discussed in reference texts on regexes to avoid ReDoS [39, 44, 45, 76]. We used their implementation provided by Davis *et al.* [29].

QOA (Quantified Overlapping Adjacency): An example of this anti-pattern is: `/\w*#?\w*/`. The two quantified `\w*` nodes overlap, and are adjacent because one can be reached from the other by skipping the optional octothorpe. From each node we walk forward looking for a reachable quantified adjacent node with an overlapping set of characters, stopping at the earliest of: a quantified overlapping node (QOA), a non-overlapping non-optional node (no QOA), or the end of the nodes (no QOA).

QOD (Quantified Overlapping Disjunction): An example of this anti-pattern is `/(\w|\d)+/`. Here we have a quantified disjunction `(/(...|...)+/)`, whose two nodes overlap in the digits, 0-9.

Star height >1: An example of this anti-pattern is `/(a+)+/`. To measure star height, we traverse the regex and maintain a counter for each layer of nested quantifier: +, *, and check if the counter reached a value higher than 1.

7.2 Ground Truth

We assessed the ground truth for whether a regex is IA or not by using one of the existing techniques intended to identify SL regexes during software maintenance: Weideman *et al.*'s [88, 89] —we avoid potential implementation bias by not implementing our own ground-truth checker, *e.g.,* applying our theory. For a given regex, Weideman *et al.*'s detector returns whether it is SL or not by analyzing its NFA. To fit our experiment, we modified it to return instead if a regex is IA or not —making it stop after looking for IDA in the NFA, skipping the search for the prefix and suffix that would make it SL

7.3 Dataset

We executed our studied anti-patterns over a large-scale dataset of real-world regexes from open source software [31]. It contains 537,806 unique regexes, extracted from 193,524 projects, written in 8 programming languages. This dataset has been used by previous studies for measuring ReDoS [31], repairing ReDoS [21, 33] and measuring general characteristics of regexes [32]. We curated the dataset for our experiment. First, we removed the 295,151 regexes that were not supported by the tool that we used to obtain ground truth (Weideman *et al.*'s [89]). Note that, while our implementation of our IA anti-patterns supported a larger percentage of the dataset (see §7.1 above), we had to discard those for which we could not collect ground truth, *i.e.,* those unsupported by Weideman *et al.*'s.

Then, we also discarded 32,413 additional regexes that were not supported by the library that we used to implement our anti-patterns (BRICS [61]), and 1,054 regexes with unicodes and posix character classes, which are not supported by our implementation. This resulted in 209,188 regexes that we analyzed in our experiment — one order of magnitude larger than the evaluation of past ReDoS-detection approaches, which analyzed 15,000-30,000 regexes *e.g.,* [56, 70, 88, 89]. 32,005 of our studied regexes were IA, according to our ground truth.

7.4 Metrics

We evaluate our studied anti-patterns using the metrics expressed in Eqt. (7.1).

- $Coverage_{ap}$ captures what ratio of all the IA regexes (irrespective of their ground-truth anti-patterns) were predicted as IA *by a given anti-pattern* (ap). When we apply $\text{Coverage}_{\text{ap}}$ to the union of all anti-patterns together, we refer to it as *Recall.*

$$\text{Coverage}_{\text{ap}} = \frac{|\text{Predicted IA}_{\text{ap}} \cap \text{Ground-truth IA}|}{|\text{Ground-truth IA}|} \tag{7.1}$$

7.5 Results

We answer two separate questions, and discuss false positives/negatives.

7.5.1 How Effective was Each of Our IA Anti-patterns at Identifying IA Regexes?

Table 7.1 shows the $\text{Coverage}_{\text{ap}}$ provided by each of our proposed IA anti-patterns. Note that the $\text{Coverage}_{\text{ap}}$ values do not add up to 100%, since some IA regexes may contain multiple anti-patterns.

We make multiple observations in this table. First, all our IA anti-patterns as a group provided very high precision (100% altogether) and very high $\text{Coverage}_{\text{ap}}$ —which is equivalent to Recall in this case (99%, making false negatives also rare).

Second, we observed wide variations in the values of $\text{Coverage}_{\text{ap}}$ for individual anti-patterns. This means that they could be further simplified and still obtain very high $\text{Coverage}_{\text{ap}}$ altogether. Somebody wanting to learn only a single anti-pattern could learn only Concat 1 and still cover 54% of IA regexes —adding Concat 2, one would cover the large majority of IA regexes, and so on.

Finally, before considering ignoring the less common (lower $\text{Coverage}_{\text{ap}}$) IA anti-patterns, one should also consider their risk. While the star anti-patterns are less common (about 6% of all IA regexes), they are riskier —our theory shows that they lead to exponential ambiguity.

7.5.2 How Effective were Our IA Anti-patterns Compared to the State-of-the-practice (SOP) Anti-patterns?

In Table 7.2, we report the results for our studied anti-pattern families. It shows that our proposed IA anti-patterns provided a substantial improvement in both precision (100% compared to 50%) and recall (99% compared to 87%) when compared to the SOP anti-

Table 7.1: **Coverage$_{\text{ap}}$** provided by our proposed IA anti-patterns. As some regexes fit multiple IA anti-patterns, the final row eliminates double-counting.

IA Anti-pattern	**# correctly flagged IA**	**Coverage$_{\text{ap}}$**
Concat 1	17,349	54%
Concat 2	12,419	39%
Concat 3	414	1%
Star 1	192	<1%
Star 2	639	2%
Star 3	1,133	4%
All anti-patterns	31,537	99%

Table 7.2: Comparison between anti-patterns

Anti-patterns	**Precision**	**Recall**
IA anti-patterns	100%	99%
SOP anti-patterns	50%	87%

patterns. After taking a closer look, we found that the main reason why our IA anti-patterns provided much higher precision than the SOP anti-patterns is that we reduced the number of false positives. For example, the *Star height > 1* anti-pattern can produce many false positives *e.g.,* the non-IA regex `/(b*c)*/` has *Star height = 2*. Similarly, the main reason why the IA anti-patterns provided higher recall is that we reduced the number of false negatives for IA regexes, *e.g.,* for regexes like `(a|b)*(ab)*` and `(a|b|ab)*`, where the SOP anti-patterns would not find any overlap between $(a|b)$ and (ab) and would not label them as IA. In contrast, our *concat 1* and *star 2* anti-patterns, respectively, would label both as IA.

We also observed that the SOP anti-patterns provided higher precision and recall in our study than in Davis *et al.*'s [29]. This difference may be caused by multiple reasons: we studied a different dataset; we applied them to identify IA regexes (Davis *et al.* applied them to identify SL regexes); and we assumed full match for unanchored regexes (*e.g.,* converting `a+` to `/^.*?a+$/`) [31]), which reveals more IA regexes in the dataset.

7.5.3 Understanding False Negatives

Finally, we also performed a deeper investigation into the root cause of the false negatives of our IA anti-patterns (the 1% of IA regexes that they did not flag as IA).

From the comparison of our IA anti-patterns with our theory chapter 6, we know that some IA regex constructions lead to IA that our anti-patterns will not capture, *i.e.,* `/((ab|a)(bc|c))*/`. However, we did not find such cases among the false negatives of our experiment.

The false negatives in our experiment were mainly regexes with constructions that were too complex for our current anti-pattern scripts to detect. While this limitation of our implementation caused a few false negatives (affecting only 1% of IA regexes), our implementation is still sound for our studied dataset —it caused no false positives.

7.6 Results Summary for Experiment 1

Experiment 1 shows a strong validation of our theory, and thus supports the proposed IA anti-patterns derived therefrom. Our IA anti-patterns correctly classified the wide majority of our studied regexes (over 99%) as true positives or true negatives, which also substantially improved the state of the practice (SOP) anti-patterns.

Chapter 8

Experiment 2: Usability

We also evaluated whether our anti-patterns help developers when used in practice to compose regexes — if developers can understand them and apply them in practice. For that goal, we recruited 20 software developers and asked them to perform 5 regex composition tasks. We asked them to use our anti-patterns to avoid introducing IA regexes in those tasks. As baseline, we also asked them to perform the same tasks using the same state-of-the-practice (SOP) anti-patterns that we studied in Experiment 1 (§7.1). We measured how many developers correctly assessed whether their composed regex was IA, for each family of anti-patterns and task.

8.1 Experiment Process

Our experiment consisted of four phases. (1) *Training phase*, where we trained the participants about basic regex syntax, how ambiguity and infinite ambiguity work in regex and some useful terminology for the study. (2) *First task phase*, where we showed participants our IA anti-patterns (as in Table 6.1) and we asked our participants to perform 5 regex composition tasks. (3) *Second task phase*, where we showed participants the SOP anti-patterns (as in §7.1) and we asked our participants to perform the same 5 regex composition tasks, in the same order. To guard against learning effects, 50% of our participants (randomly selected) applied first the IA anti-patterns first, and second the SOP anti-patterns, and the other 50% applied them in the opposite order. All participants applied both sets of anti-patterns for all 5 tasks. (4) *Concluding questions phase*, where we asked the participants some concluding questions.

For each regex composition task, we asked the participants to evaluate their regex drafts to see if they were IA or not, using the assigned set of anti-patterns. We emphasized that developers should apply the given anti-patterns, and not their opinion. We also explained that they should not think that one set of anti-patterns was more correct than the other one —so that they applied the anti-patterns, regardless from their perception of their correctness. To learn if the anti-patterns were usable in a reasonable time-frame, we gave participants a limit of 5 minutes to complete each task. We also asked them to think out loud. One author of this work was present during all the human subject experiments to answer clarifying questions.

8.2 Participant Recruitment

The target audience for this experiment consisted of practitioners with professional experience in software development, with knowledge about writing regexes (knows what repetition operators like $*$ and $+$ do). To recruit and screen for participants with these characteristics, we distributed a survey. We obtained approval from our institution's ethics board and posted the survey on Twitter, Reddit (r/regex) and our institution's mailing lists.

After screening our survey respondents for the criteria above, we performed our experiment with 21 eligible participants. After the experiment was performed, we discarded one participant, who composed incorrect regexes —that did not match any of the examples provided in the specification— for 70% of the study tasks. As a result, we analyzed the performance of 20 participants in this experiment. We compensated each participant with a $15 Ama-

Table 8.1: Our studied regex composition tasks.

Task	**Description**	**Possible solution**
1	Write one regex that matches one or more ‘b’ followed by a single ‘c’: Example matching strings: bc, bbc, bbbbc, bbbbbc, bbbbbbbbbbbbbbbbbbbbbc	non-IA: `b+c` IA: `(b+)+c`
2	Write one regex that matches one or more repetitions of the following: one or more ‘b’ followed by a single ‘c’: Example matching strings: bcbc, bbcbbcbbc, bbbbbcbbbbbc, bbbbbbbbbbbbbbbbbbbbbcbbbbbbbbbbbbbbbbbbbbc	non-IA: `(b+c)+` IA: `((b+)+c)+`
3	Write a regex to match one or more ‘a’ or ‘b’, followed by one or more repetitions of ‘ab’ Example matching strings:aab, bab, aaab, aaaaab, bab, bbbab, aaaabababab, bbbbabababababab	non-IA: `(a+|b+)(ab)+` IA: `(a|b)+(ab)+`
4	Write a regex to match one or more occurrences of the strings ‘a’, ‘b’, or ‘ab’. Example matching strings: aaaaaaaaaa, bbbbbbbbbbbb, ababababababababab	non-IA: `a+|b+|(ab)+` IA: `(a|b|ab)+`
5	Write one regex that matches one or more ‘a’ followed by an optional ‘b’ followed by one or more ‘a’. Example matching strings: aaaabaa, aaaaa, abaaaa	non-IA: `a+|a+ba+` IA: `(a+b?a+)`

zon gift card for participating in our study. Our participants had a median 1-2 years of programming experience and median intermediate level of regex expertise (they understood non-simple features, like non-greedy quantifiers, and character classes).

8.3 Studied Techniques

In this human-subjects experiment, we aim to evaluate the effectivenes provided by our IA anti-patterns, in comparison with existing techniques that were proposed for the same context *i.e.,* to be applied manually by humans, at regex-composition time. Thus, we evaluated our IA anti-patterns and the SOP anti-patterns, as described in §7.1.

8.4 Tasks

We list the tasks that we asked our participants to perform in Table 8.1. We designed these tasks so that participants could perform them in a bounded amount of time (5 minutes), while still being representative of real-world regex composition. The draft solutions that we anticipated for our tasks (and that our participants composed) had similar complexity to typical real-world regexes according to [32]. For example, drafts had length 6-11 (median regex length in Java: 15) and used 2-3 operators (Java: median 3).

Task 1 was an easy task, as warm-up, to get participants familiar with the structure of the experiment. We designed the remaining 4 tasks to evaluate 4 specific scenarios. Tasks 2, 3, and 4 targeted limitations that we identified in each of the three SOP anti-patterns, to learn if our IA anti-patterns are more effective in those scenarios. Task 5 targets a scenario in which the SOP anti-patterns are not limited, to learn if our IA anti-patterns are not worse in such scenario. We expected both sets of anti-patterns to perform equally in tasks 1 and 5, and our IA anti-patterns to be more effective in tasks 2, 3, and 4. For task 2, we expected *Star height* > 1 to flag the participant's composed regex as IA, even when it is not. For task 3, we expected *QOA* to not flag the participant's composed regex as IA, even when it is. For task 4, we expected *QOD* to not flag the participant's composed regex as IA, even when it is.

8.5 Ground Truth

We used the same ground truth as in our previous experiment (§7.2) to label the regexes composed by our participants as IA or not.

Table 8.2: Accuracy: ratio of participants correctly using each anti-pattern set to identify if their composed regex was IA.

	SOP first, IA after ($N = 10$)		**IA first, SOP after** ($N = 10$)		**All orders** ($N = 20$)	
Task	**SOP Accuracy**	**IA Accuracy**	**SOP Accuracy**	**IA Accuracy**	**SOP Accuracy**	**IA Accuracy**
1	100%	100%	100%	100%	100%	100%
2	10%	100%	0%	100%	5%	100%
3	20%	100%	20%	100%	20%	100%
4	30%	100%	20%	100%	25%	100%
5	100%	100%	100%	100%	100%	100%
All	52%	100%	48%	100%	50%	100%

8.6 Metrics

We measured how successfully our studied anti-patterns helped our study participants to correctly identify whether their composed regexes were IA. For each studied task and possible anti-pattern ordering (SOP first, or IA first), we measured the **accuracy** of each anti-pattern set, *i.e.,* the ratio of participants that could correctly identify whether their composed regex was IA or not when using them.

8.7 Results

We report the results of our human-subjects experiment in Table 8.2. We calculated accuracy as the ratio of participants that correctly identified whether their composed regex was IA. In more detail:

For task 1, all participants (for all anti-pattern sets) composed a non-IA regex, and correctly identified it as non-IA. We expected this result, since this was the warm-up task.

For task 2, all participants when using SOP composed a non-IA regex, but only 1/20 correctly identified it as non-IA. All participants when using our IA anti-patterns composed a non-IA regex, and all correctly identified it as non-IA. In this task, the SOP anti-patterns misled most participants to think that their composed regex was IA, when it was not.

For task 3, most participants when using SOP (15/20) composed an IA regex, but only 1/15 correctly identified it as IA. The remaining (5/20) participants when using SOP composed a non-IA regex, but only 3/5 correctly identified it as non-IA. Then, most participants when using our IA anti-patterns (16/20) composed an IA regex, and all correctly identified it as IA. The remaining (4/20) participants when using our IA anti-patterns composed a non-IA regex, and all correctly identified it as non-IA. In this task, the SOP anti-patterns misled most participants to think that their composed regex was non-IA, when it was in fact IA.

For task 4, most participants when using SOP (13/20) composed an IA regex, but none correctly identified it as IA. The remaining (7/20) participants when using SOP composed a non-IA regex, but only 5/7 correctly identified it as non-IA. Then, most participants when using our IA anti-patterns (15/20) composed an IA regex, and all correctly identified it as IA. The remaining (5/20) participants when using our IA anti-patterns composed a non-IA regex, and all correctly identified it as non-IA. In this task, the SOP anti-patterns misled most participants to think that their composed regex was non-IA, when it was in fact IA.

For task 5, all participants (for all anti-patterns) composed an IA regex, and correctly identified it as IA. We also expected this result, since this was the task in which we evaluated non-inferiority, *i.e.,* whether our IA anti-patterns would perform well in a scenario where the SOP anti-patterns were not limited.

We make multiple observations from the results of this experiment. First, our proposed IA anti-patterns allowed 100% of participants to correctly identify whether their composed regex was IA, for all tasks, and for all orderings. This is not surprising given that they are derived from our theory, and the high precision and recall that they provided in chapter 7. However, this result also shows that practitioners can apply them with high success in regex composition tasks.

Second, the SOP anti-patterns showed their limitations in tasks 2, 3, and 4, as we expected (§8.4) —and they were very similar irrespective of the ordering. For tasks 1 and 5, both the SOP and IA anti-patterns allowed 100% of practitioners to correctly identify if the composed regex was IA, also irrespective of ordering. This shows that our IA anti-patterns are much more effective than the SOP ones in the situations when they are limited, and as good as them when they are not.

We also tested for statistical significance applying the Wilcoxon signed rank test, because our observations were not normally distributed and paired. We measured the difference in the correctness of the IA assessments that participants produced when using our IA anti-patterns in comparison with when they used the SOP anti-patterns. We tested this difference separately for each ordering, and also for all orders together. For all these tests, the differences observed were statistically significant ($p < .00001$)

Finally, in the concluding questions, we asked our studied participants which set of anti-patterns was easier to understand and apply. 15 out 20 participants said our IA anti-patterns were easier, 3 participants preferred the SOP anti-patterns and 2 participants were undecided.

8.8 Results Summary for Experiment 2

Practitioners were able to apply our IA anti-patterns in practice to compose regexes, and accurately understand whether they were IA or not (in 100% of our observations). This substantially outperformed the application of existing SOP anti-patterns, with which for some tasks only 5-25% practitioners could accurately determine whether their composed regex was IA or not.

Chapter 9

Discussion

9.1 Defense in Depth

Our proposed theory and anti-patterns complement the existing multi-layer defense against ReDoS (see Figure 1.1). The existing approaches target the software maintenance (see §3.3 "Coping with SL Regexes") and deployment stages (see §3.2 "Changing the Infrastructure").

We propose a novel approach (a theory and anti-patterns) to target the regex composition stage. Our approach complements the existing multi-layer defense strategy with a different goal: helping developers avoid introducing SL regexes in the first place —allowing them to manually assess whether their regexes are IA as they are composing them. As complement, developers can still continue to apply techniques later in software maintenance to automatically identify and repair SL regexes that have accidentally slipped through our proposed first layer of protection.

This multi-layer approach to software security is commonly known as *defense in depth*, or the "*Swiss cheese*" approach [7, 8, 51].

There are two major benefits to applying multiple complementary layers of defense against security problems. First, the strengths of a layer may counterbalance the limitations of others, *e.g.,* coding guidelines may avoid problems that are hard to detect statically. Second, one layer may reach developers that decided to not use others. The high-profile ReDoS outages [37, 46] involved regexes that could have been caught by static analysis tools — but these organizations were not using the right tools.

Thus, the defense in depth approach is a general engineering principle, applied to prevent many threats in software engineering (cybersecurity and otherwise). Most industry practitioners (86.2%) complement automated tools with manual assessments and/or code review for secure coding [41]. Security experts recommend that developers "think secure from the beginning" [11], and follow a multi-method strategy of: static analysis, dynamic analysis, penetration testing, and manual inspection [79]. More generally, software developers also apply many layers of defense against design flaws: they learn and apply design patterns [40] to avoid them at composition time; apply program analyses to detect them statically, *e.g.,* high coupling [3]; and additionally look for them in code review [57].

9.2 Tools vs. Rules

Two families of approaches have been proposed to address ReDoS during the software maintenance stage — to automatically identify and repair SL regexes. Some of these techniques could be adapted for usage during regex composition in two ways.

First, developers could apply the tools' static analyses in their head during composition, reasoning about their regex drafts in their NFA or evaluation-tree modalities. However, many developers may only understand (or prefer to only use) the regex modality [13].

Second, developers could run these techniques in the background, continuously automatically analyzing their regex drafts. No study of this approach has been conducted. Johnson *et al.* imply that this scheme may be counterproductive for many engineers, who would find it too invasive [50] —particularly when they interrupt their workflow, produce hard-to-understand results, or provide no suggestions to fix the found problems. Furthermore, developers may be particularly unlikely to adopt verification tools for regexes. Recent work found that regexes are poorly tested —software engineers cannot detect many regex regressions via unit test suites [83].

Our proposed anti-patterns (based on the regex modality) complement such strategies, to now reach a wider population of developers: those that only understand (or prefer to use) the regex modality, and those that decide to not use existing static analysis tools during regex composition.

9.3 Long-term Utility of our Anti-patterns

Several researchers have proposed ReDoS elimination by changing the regex engine algorithm [25, 33, 48, 81]. These schemes would permanently eliminate ReDoS Condition 1, obviating all other ReDoS aids, including the one proposed here. However, the nature of legacy software makes it likely that ReDoS vulnerabilities will persist. Engineers working on new web services can take advantage of these schemes, but existing web services will remain vulnerable until the application is ported to a safe regex engine. Davis *et al.* showed that this task is not trivial [31]. Furthermore, relying only on this layer of defense would imply that IA regexes stay in the source code as a latent threat that can be triggered as soon as the current project stops using these defense mechanisms, or when they are copied and pasted into a different project that does not use them.

Chapter 10

Threats to Validity

10.0.1 Internal Validity

To guard internal validity, we carefully tested the implementation scripts of our anti-patterns over small samples of the dataset. We also curated our studied dataset to prepare it for our experiments. To the extent possible, we applied existing implementations of tools in our evaluation, *e.g.,* the SOP anti-patterns by Davis *et al.* [29] and Weidemann *et al.*'s SL detector [89], to avoid errors introduced by implementing them ourselves. Finally, in our human-subjects study, we studied both orders of anti-pattern application to the same extent, assigning them to participants randomly.

10.0.2 External Validity

To increase external validity, we selected a popular dataset for our experiments [31]. We also carefully screened the participants of our human-subjects experiment, so that it reflected the target audience for our anti-patterns —participants with professional experience and regular expression writing experience. We also studied a diverse set of tasks in our human-subjects experiment, to understand various situations in regex composition. Finally, we also applied multiple evaluation methods to our IA anti-patterns, that complement each other. Experiment 1 shows that our anti-patterns were comprehensive, and experiment 2 shows that practitioners can apply them successfully.

Chapter 11

Conclusions

In this paper, we proposed a novel theory of regex infinite ambiguity (IA) to characterize regexes that can become vulnerable to ReDoS. We also proposed a set of IA anti-patterns, derived from our theory, that developers can apply at regex composition time, to avoid inserting regexes vulnerable to ReDoS in their source code.

We evaluated our IA anti-patterns in comparison with the state-of-the-practice (SOP) anti-patterns, to understand both their comprehensiveness, and their usability. Our evaluation showed that our proposed IA anti-patterns provide much higher precision and recall detecting real-world IA regexes, and provide higher accuracy when used in practice. In the future, we plan to apply our thorough methodology (developing a theory and anti-patterns) to also strongly address other similar important security problems (*e.g.,* in GraphQL).

Our contributions have significance for secure software engineering. They will help reduce the impact of ReDoS: an important security vulnerability that affects hundreds of popular websites [73]. Our theory is a foundational contribution to characterize an important concept (regex infinite ambiguity) in a modality that was not addressed before (the regex), and which is the more often preferred by developers [13]. Our anti-patterns will enable developers to address an important gap in the protection against ReDoS: having a highly effective defense mechanism at regex composition time.

Bibliography

[1] antlr-pcre. https://web.archive.org/web/20210826063830/https://github.com/bkiers/pcre-parser.

[2] Owasp modsecurity core rule set,. https://coreruleset.org/.

[3] Codemr static analysis tools,. https://www.codemr.co.uk/.

[4] Google cloud devops tech: Shifting left on security. https://cloud.google.com/architecture/devops/devops-tech-shifting-left-on-security, .

[5] A modern shift-left security approach. https://www.forbes.com/sites/forbestechcouncil/2021/01/04/a-modern-shift-left-security-approach/, .

[6] To improve devops and security, the time has come to "shift left". https://www.securityroundtable.org/to-improve-devops-and-security-the-time-has-come-to-shift-left/, .

[7] Swiss cheese model. https://en.wikipedia.org/wiki/Swiss_cheese_model, .

[8] The swiss cheese model: Designing to reduce catastrophic losses. https://www.engineeringforhumans.com/systems-engineering/the-swiss-cheese-model-designing-to-reduce-catastrophic-losses/, .

[9] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools*. 2020.

[10] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata. In *International Conference on Developments in Language Theory*, pages 108–120. Springer, 2008.

[11] Hala Assal and Sonia Chiasson. 'think secure from the beginning' a survey with software developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–13, 2019.

[12] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[13] Gina R Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T Stolee. Exploring tools and strategies used during regular expression composition tasks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 197–208. IEEE, 2019.

[14] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. Runtime recovery of web applications under zero-day redos attacks. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1575–1588. IEEE, 2021.

[15] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 387–401, 2008. ISBN 9780769531687. doi: 10.1109/SP.2008.22.

[16] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. *EPTCS: Automata and Formal Languages 2014*, 151:109–123, 2014. ISSN 20752180. doi: 10.4204/EPTCS.151.7. URL https://arxiv.org/pdf/1405.5599.pdf.

[17] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. *arXiv preprint arXiv:2002.03416*, 2020.

[18] Claus Brabrand and Jakob G Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 243–254, 2010.

[19] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[20] Carl Chapman and Kathryn T Stolee. Exploring regular expression usage and context in Python. *International Symposium on Software Testing and Analysis (ISSTA)*, 2016. doi: 10.1145/2931037.2931073.

[21] Nariyoshi Chida and Tachio Terauchi. Automatic repair of vulnerable regular expressions. *arXiv preprint arXiv:2010.12450*, 2020.

[22] Laura Chiticariu, Vivian Chu, Sajib Dasgupta, Thilo W. Goetz, Howard Ho, Rajasekar Krishnamurthy, Alexander Lang, Yunyao Li, Bin Liu, Sriram Raghavan, Frederick R. Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. The systemt ide: An integrated development environment for information extraction rules. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1291–1294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989479. URL http://doi.acm.org/10.1145/1989323.1989479.

[23] Miles Claver, Jordan Schmerge, Jackson Garner, Jake Vossen, and Jedidiah McClurg.

Regis: Regular expression simplification via rewrite-guided synthesis. *arXiv preprint arXiv:2104.12039*, 2021.

[24] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, page 1280–1287, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349208. doi: 10.1145/3071178.3071196. URL https://doi.org/10.1145/3071178.3071196.

[25] Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), 2007. URL https://swtch.com/~rsc/regexp/regexp1.html.

[26] Russ Cox. Regular Expression Matching in the Wild, 2010. URL https://swtch.com/~rsc/regexp/regexp3.html.

[27] Scott Crosby. Denial of service through regular expressions. *USENIX Security work in progress report*, 2003.

[28] Scott A Crosby and Dan S Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security*, 2003.

[29] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018. ISBN 9781450355735.

[30] James C Davis, Eric R Williamson, and Dongyoon Lee. A sense of time for javascript and

node. js: First-class timeouts as a cure for event handler poisoning. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 343–359, 2018.

[31] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019. ISBN 9781450355728. doi: 10.1145/3338906.3338909.

[32] James C Davis, Daniel Moyer, Ayaan M Kazerouni, and Dongyoon Lee. Testing Regex Generalizability And Its Implications: A Large-Scale Many-Language Measurement Study. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[33] James C Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (redos). In *2021 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA*, pages 543–559, 2021.

[34] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 693–708, 2019.

[35] The Rust Project Developers. regex - rust. https://docs.rs/regex/1.1.0/regex/.

[36] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

[37] Stack Exchange. Outage postmortem. http://web.archive.org/

web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[38] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, 2004.

[39] Jeffrey EF Friedl. *Mastering regular expressions*. O'Reilly Media, Inc., 2002.

[40] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Elements of reusable object-oriented software*, volume 99. Addison-Wesley Reading, Massachusetts, 1995.

[41] Tiago Espinha Gasiba, Ulrike Lechner, Maria Pinto-Albuquerque, and Daniel Mendez. Is secure coding education in the industry needed? an investigation through a large scale survey. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 241–252. IEEE, 2021.

[42] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. Hare: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016. doi: 10.1109/MICRO.2016.7783747.

[43] Google. regexp - go. https://golang.org/pkg/regexp/.

[44] Jan Goyvaerts. *Regular Expressions: The Complete Tutorial*. Lulu Press, 2006.

[45] Jan Goyvaerts and Steven Levithan. *Regular expressions cookbook*. O'reilly, 2012.

[46] Graham-Cumming, John. Details of the cloudflare outage on july 2, 2019. https://web.archive.org/web/20190712160002/https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[47] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[48] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. Succinct determinisation of counting automata via sphere construction. In *Asian Symposium on Programming Languages and Systems*, pages 468–489. Springer, 2019.

[49] Jamie Jennings. Rosie pattern language (rpl). https://gitlab.com/rosie-pattern-language/rosie.

[50] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[51] Faouzi Kamoun and Mathew Nicho. Human and organizational factors of healthcare data breaches: The swiss cheese model of data breach causation and prevention. *International Journal of Healthcare Information Systems and Informatics (IJHISI)*, 9(1): 42–60, 2014.

[52] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. *Network and System Security*, 7873:35–148, 2013. ISSN 03029743. doi: 10.1007/978-3-642-38631-2{_}11.

[53] SC Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[54] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. Flashregex: deducing anti-redos regexes from examples. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 659–671. IEEE, 2020.

[55] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1613715.1613719.

[56] Y. Liu, M. Zhang, and W. Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1468–1484, Los Alamitos, CA, USA, may 2021. IEEE Computer Society. doi: 10.1109/SP40001.2021.00062. URL https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00062.

[57] Mika V Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2008.

[58] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. Regulator: Dynamic analysis to detect redos. 2022.

[59] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 393–410, 2018.

[60] Louis G Michael IV, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. Regexes are Hard : Decision-making, Difficulties, and Risks in Programming Regular Expressions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[61] Anders Møller. dk. brics. automaton–finite-state automata and regular expressions for java, 2010.

[62] Yannic Noller, Rody Kersten, and Corina S Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 322–332, 2018.

[63] Nga Lam Or, Xing Wang, and Derek Pao. Memory-based hardware architectures to detect clamav virus signatures with restricted regular expression features. *IEEE Transactions on Computers*, 65(4):1225–1238, 2016. doi: 10.1109/TC.2015.2439274.

[64] Theoolos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Computer and Communications Security (CCS)*, 2017. doi: 10.1145/3133956.3134073. URL https://arxiv.org/pdf/1708.08437.pdf.

[65] Asiri Rathnayake and Hayo Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. *CoRR*, 2014.

[66] James Reason. The contribution of latent human failures to the breakdown of complex systems. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 327(1241):475–484, 1990.

[67] James Reason. Human error: models and management. *Bmj*, 320(7237):768–770, 2000.

[68] Alex Roichman and Adar Weidman. VAC - ReDoS: Regular Expression Denial Of Service. *Open Web Application Security Project (OWASP)*, 2009.

[69] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–378. Springer, 2019.

[70] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Automated Software Engineering (ASE)*, 2018. ISBN 9781450359375.

[71] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

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

[73] Cristian-Alexandru Staicu and Michael Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium (USENIX Security)*, 2018. URL `https://www.npmjs.com/package/safe-regexhttp://mp.binaervarianz.de/ReDoS_TR_Dec2017.pdf`.

[74] Richard Edwin Stearns and Harry B Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

[75] Martin R Stytz. Considering defense in depth for software applications. *IEEE Security & Privacy*, 2(1):72–75, 2004.

[76] substack and Davis. safe-regex. `https://www.npmjs.com/package/safe-regex`, 2013.

[77] Satoshi Sugiyama and Yasuhiko Minamide. Checking Time Linearity of Regular Expression Matching Based on Backtracking. *Information and Media Technologies*, 9(3): 222–232, 2014.

[78] Martin Sulzmann and Kenny Zhuo Ming Lu. Derivative-Based Diagnosis of Regular Expression Ambiguity. *International Journal of Foundations of Computer Science*, 28 (5):543–561, 4 2017. ISSN 01290541. doi: 10.1142/S0129054117400068. URL http://arxiv.org/abs/1604.06644.

[79] Tyler W Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[80] Ken Thompson. Regular Expression Search Algorithm. *Communications of the ACM (CACM)*, 1968.

[81] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[82] Brink Van Der Merwe, Nicolaas Weideman, and Martin Berglund. Turning Evil Regexes Harmless. In *SAICSIT*, 2017. ISBN 9781450352505. doi: 10.1145/3129416.3129440. URL https://doi.org/10.1145/3129416.3129440.

[83] Peipei Wang and Kathryn T Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 668–678, 2018.

[84] Peipei Wang, Gina R Bai, and Kathryn T Stolee. Exploring regular expression evolution. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–513. IEEE, 2019.

[85] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2008. ISBN 9781605580791. doi: 10.1145/1368088.1368112.

[86] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theoretical Computer Science*, 88(2):325–349, 1991.

[87] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 213–223, 2018.

[88] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9705, pages 322–334, 2016. ISBN 9783319409450. doi: 10.1007/978-3-319-40946-7{_}27.

[89] Nicolaas Hendrik Weideman. *Static Analysis of Regular Expressions*. PhD thesis, Stellenbosch University, 2017.

[90] Valentin Wustholz, Oswaldo Olivo, Marijn J H Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2017.

Appendices

Appendix A

Proofs of Theorems

A.1 Definitions

We define the operators that we use in theorems 2 and 3.

A.1.1 $\bowtie$

Brabrand & Thomsen [18] introduced an overlap operator, $\bowtie$, between two languages $L(R_1)$ and $L(R_2)$. The set $L(R1) \bowtie L(R2)$ contains the ambiguity-inducing strings that can be parsed in multiple ways across $L(R1)$ and $L(R2)$. More formally, with $X = L(R_1)$ and $Y = L(R_2)$,

$$X \bowtie Y = \{xay \mid x, y \in \Sigma^* \wedge a \in \Sigma^+ \wedge x, xa \in X \wedge ay, y \in Y\}$$

A.1.2 Ω

Brabrand & Thomsen use Møller's BRICS library [61] for the implementation of their theorems and actually rely on what we call the Møller overlap operator, Ω. We use this operator in our theorems. The Møller overlap operator describes only the ambiguous core "a":

$$X \ \Omega \ Y = \{a \mid a \in \Sigma^{+} \wedge \exists \ x, y \in \Sigma^{*} s.t. \ x, xa \in X \wedge ay, y \in Y\}$$

A.2 Assumptions

In our theorems and proofs, we assume that we can convert regexes to their equivalent, ambiguity-preserving, ϵ-free NFAs [86, 89] (with no ϵ-loops).

A.3 Theorems & proofs

Brabrand & Thomsen's Theorem 0 [18] provides the conditions for *unambiguity*. Our proofs consider the effect of negating the unambiguity condition, and distinguish the conditions that lead to finite or infinite ambiguity.

A.3.1 Theorem 1: Ambiguity of Alternation

Given unambiguous regexes R_1 and R_2,

(a) *$R_1|R_2$ is finitely ambiguous iff $L(R_1) \cap L(R_2) \neq \phi$.*

(b) *$R_1|R_2$ cannot be infinitely ambiguous.*

The components of Theorem 1 follow from proposition A.1.

Lemma A.1. *Given unambiguous R_1 and R_2, if $R_1|R_2$ is ambiguous it is always finitely ambiguous.*

Proof. Given a string s there are four outcomes when it is matched against $R_1|R_2$ — s may be matched by R_1, by R_2, by both, or by neither. In any case, since R_1 and R_2 are unambiguous, there are at most two ways for $R_1|R_2$ to match s.

$\square$

A.3.2 Theorem 2: Ambiguity of Concatenation

Suppose unambiguous regexes R_1 and R_2, and that $L(R_1) \bowtie L(R_2) \neq \phi$ (so $R_1 \cdot R_2$ is ambiguous by Theorem 0). Then:

(a) *$R_1 \cdot R_2$ is infinitely ambiguous iff $L(R1)$ contains the language of a regex $BC*D$ and $L(R2)$ contains the language of a regex $EF*G$, where $\epsilon \notin L(C) \wedge \epsilon \notin L(F) \wedge L(C) \cap L(F) \cap L(DE) \neq \phi$.*

(b) *Otherwise, $R_1 \cdot R_2$ must be finitely ambiguous.*

2(a) is an iff so we need to prove:

$\Longrightarrow$: If $L(R1)$ contains the language of a regex $BC*D$ and $L(R2)$ contains the language of a regex $EF*G$, where $\epsilon \notin L(C) \wedge \epsilon \notin L(F) \wedge L(C) \cap L(F) \cap L(DE) \neq \phi$, then $R_1 \cdot R_2$ is infinitely ambiguous.

Proof. Consider a string $q = bc^m d \in L(BC*D)$ where $b, d \in \Sigma^*$, $c \in \Sigma^+$, $b \in L(B)$, $c \in L(C)$, and $d \in L(D)$. By hypothesis, $L(BC*D) \subset L(R1)$, so $q \in L(R1)$. Similarly, consider another string $r = ef^n g \in L(EF*G)$ where $e, g \in \Sigma^*$, $f \in \Sigma^+$, $e \in L(E)$, $f \in L(F)$, and $g \in L(G)$. By hypothesis, $L(EF*G) \subset L(R2)$, so $r \in L(R2)$. As $L(C) \cap L(F) \cap L(DE) \neq \phi$, suppose $c = f = de$.

Consider the new string $p = qr = bc^m def^n g \in L(R1) \cdot L(R2) = L(R1 \cdot R2)$. In other words, $R_1 \cdot R_2$ should include the following NFA accepting p.

$$\begin{tikzcd} {} \arrow[r] & \cdots \arrow[r, "b"] & v_1 \arrow[loop above, "c"] \arrow[r, "d"] & v_2 \arrow[r, "e"] & v_3 \arrow[loop above, "f"] \arrow[r, "g"] & \cdots \end{tikzcd}$$

For $m = 2$ and $n = 2$, $p = bccdeffg$. There are 5 ways to match. Ignoring prefix b and suffix g, the five cases to match the middle $ccdeff$ are:

- $v_1 \rightarrow^{c} v_1 \rightarrow^{c} v_1 \rightarrow^{(de=c)} v_1 \rightarrow^{(f=c)} v_1 \rightarrow^{(f=de)} v_3$
- $v_1 \rightarrow^{c} v_1 \rightarrow^{c} v_1 \rightarrow^{(de=c)} v_1 \rightarrow^{(f=de)} v_3 \rightarrow^{f} v_3$
- $v_1 \rightarrow^{c} v_1 \rightarrow^{c} v_1 \rightarrow^{d} v_2 \rightarrow^{e} v_3 \rightarrow^{f} v_3 \rightarrow^{f} v_3$
- $v_1 \rightarrow^{c} v_1 \rightarrow^{(c=de)} v_3 \rightarrow^{(de=f)} v_3 \rightarrow^{f} v_3 \rightarrow^{f} v_3$
- $v_1 \rightarrow^{(c=de)} v_3 \rightarrow^{c=f} v_3 \rightarrow^{(de=f)} v_3 \rightarrow^{f} v_3 \rightarrow^{f} v_3$

where the superscript of an arrow represents the (input observed = path taken) pair.

The degree of ambiguity grows for each larger m and n (a function of input length). Therefore, $R_1 \cdot R_2$ is infinitely ambiguous.

$\square$

$\Longleftarrow$: If $R_1 \cdot R_2$ is infinitely ambiguous, then $L(R1)$ contains the language of a regex $BC*D$ and $L(R2)$ contains the language of a regex $EF*G$, where $\epsilon \notin L(C) \wedge \epsilon \notin L(F) \wedge L(C) \cap L(F) \cap L(DE) \neq \phi$.

Proof. We will reason over an equivalent, ambiguity-preserving, ϵ-free NFA [86]. The NFA of an infinitely ambiguous regex should include either a Polynomial or an Exponential Degree of Ambiguity (PDA, EDA) section [86], as shown in Figure 2.1.

We first show that if $R_1 \cdot R_2$ is infinitely ambiguous, then the NFA of $R_1 \cdot R_2$ must contain a

PDA (Figure 2.1(a)). R_1 and R_2 are unambiguous, so none of them should have a full EDA. Concatenating two regexes $R_1 \cdot R_2$ cannot create a new self loop of EDA. Thus, $R_1 \cdot R_2$ must contain a PDA.

Consider the two nodes p with the loop π_1 and q with the loop π_3 in Figure 2.1(a). As R_1 and R_2 are unambiguous, neither R_1 nor R_2 can include both nodes p and q — because then they would be infinitely ambiguous (not unambiguous). Therefore, $R1_1$ and R_2 each should have a part of PDA; and the partition will appear somewhere along the path π_2 as the loops π_1 and π_3 cannot be newly introduced via concatenation.

Each partition of PDA consists of a prefix, a loop, and a suffix, which can be mapped to a regex of the form `PQ*R`. As a PDA is a part of the whole NFA, more generally, we can say that (1) $L(R1)$ contains the language of a regex $BC*D$ and (2) $L(R2)$ contains the language of a regex $EF*G$: *i.e.,* $L(BC*D) \subset L(R1)$ and $L(EF*G) \subset L(R2)$.

After concatenation, the full PDA can be represented by a language of the form $BC*DEF*G$, where $C*$ is mapped to the first loop π_1, DE to the path π_2, and $F*$ to the second loop π_3. Let s be the string that meets the PDA path conditions: $label(\pi_1) = label(\pi_2) = label(\pi_3)$. Then, $s \in L(C)$ (by $label(\pi_1)$, $s \in L(DE)$, and $s \in L(F)$ And thus $\epsilon \notin L(C) \wedge \epsilon \notin L(F) \wedge L(C) \cap L(F) \cap L(DE) \neq \phi$.

□

Theorem 2(b) follows from elimination with Theorem 0.

A.3.3 Theorem 3: Ambiguity of Star

Given unambiguous regex R,

(a) *$R*$ is infinitely ambiguous iff $\epsilon \in L(R) \vee L(R)\ \Omega\ L(R*) \neq \phi$.*

(b) *$R*$ cannot be finitely ambiguous.*

The components of Theorem 3 follow from proposition A.2.

Lemma A.2. *if R^* is ambiguous, it is always infinitely ambiguous.*

Proof. We prove this by induction. From the contrapositive of Theorem 0(c), if $R*$ is ambiguous, $L(R) \bowtie L(R*) \neq \phi$. There exists an input string $s = xay$ such that 1) $x, y \in \Sigma^*$, 2) $a \in \Sigma^+$, 3) $x, xa \in L(R)$, 4) $y, ay \in L(R*)$. In other words, there are at least two ways to parse s.

Now consider $ss = (xay)(xay)$. Let $x' = x, a' = a, y' = yxay$ then, $ss = x'a'y'$. Then the following conditions are true: 1) $x', y' \in \Sigma^*$, 2) $a' \in \Sigma^*$, 3) $x', x'a' \in L(R)$, and 4) $y', a'y' \in L(R * RR*) \subset L(R*)$. Now, for each xay there are at least 2 accepting paths. Therefore, for ss there are at least 4 accepting paths. The degree of ambiguity grows for each additional concatenation of an s. Therefore, $R*$ is infinitely ambiguous.

□